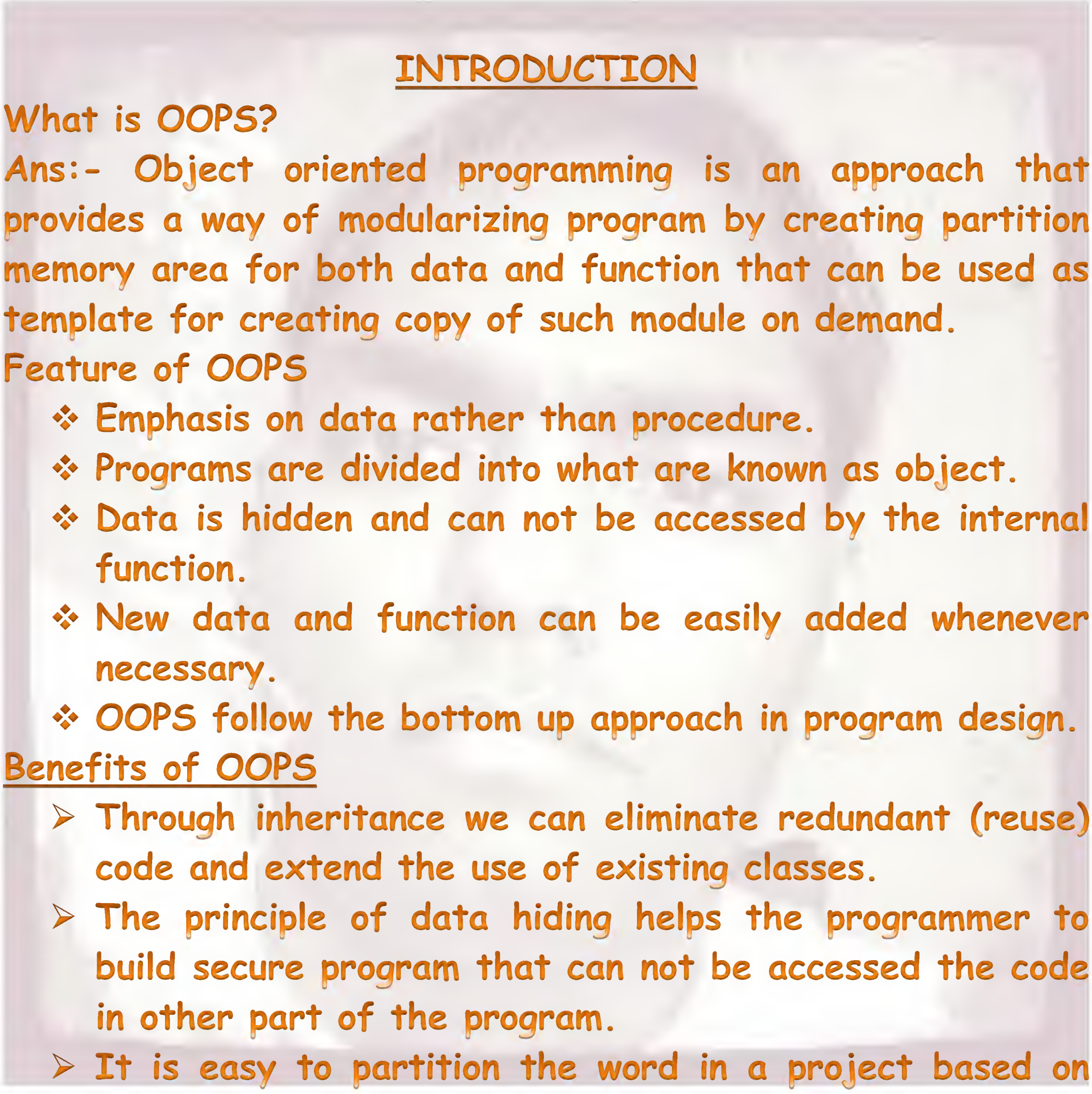
OOPS By Mritunjay



INTRODUCTION

What is OOPS?

Ans:- Object oriented programming is an approach that provides a way of modularizing program by creating partition memory area for both data and function that can be used as template for creating copy of such module on demand.

Feature of OOPS

- ❖ Emphasis on data rather than procedure.
- ❖ Programs are divided into what are known as object.
- ❖ Data is hidden and can not be accessed by the internal function.
- ❖ New data and function can be easily added whenever necessary.
- ❖ OOPS follow the bottom up approach in program design.

Benefits of OOPS

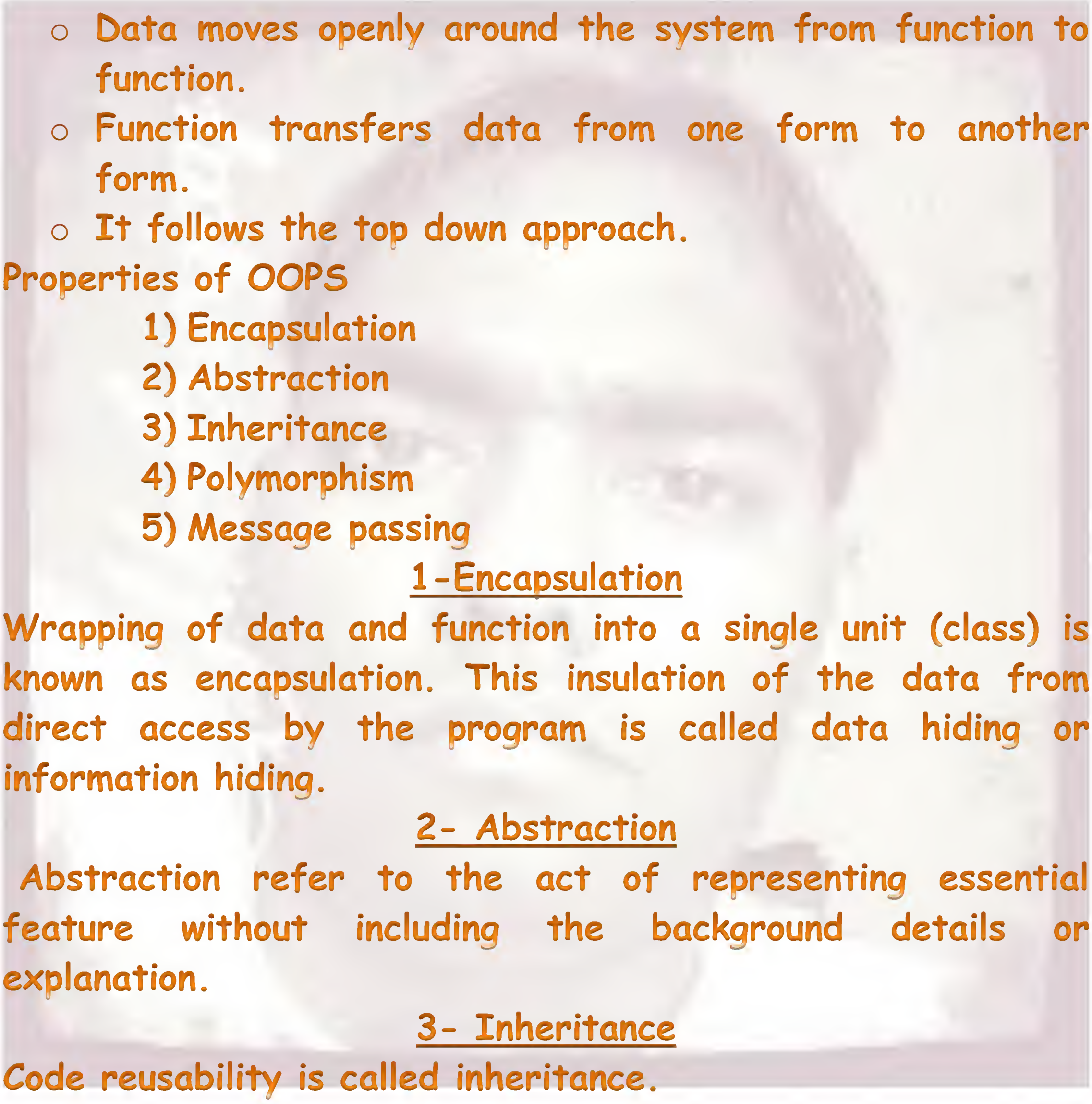
- ➢ Through inheritance we can eliminate redundant (reuse) code and extend the use of existing classes.
- ➢ The principle of data hiding helps the programmer to build secure program that can not be accessed the code in other part of the program.
- ➢ It is easy to partition the word in a project based on object.
- ➢ OOPS can be easily upgraded from small to large.
- ➢ Software complexity can be managed easily.

Procedure Oriented Language

In the procedure oriented language, the problem is viewed as a sequence of things to be done such as reading, calculating and printing.

Characteristics of procedure Oriented Language

- Emphasis is on doing things.
- Large programs are divided into smaller programs known as functions.



- Data moves openly around the system from function to function.
- Function transfers data from one form to another form.
- It follows the top down approach.

Properties of OOPS

1) Encapsulation
2) Abstraction
3) Inheritance
4) Polymorphism
5) Message passing

1-Encapsulation

Wrapping of data and function into a single unit (class) is known as encapsulation. This insulation of the data from direct access by the program is called data hiding or information hiding.

2- Abstraction

Abstraction refer to the act of representing essential feature without including the background details or explanation.

3- Inheritance

Code reusability is called inheritance.

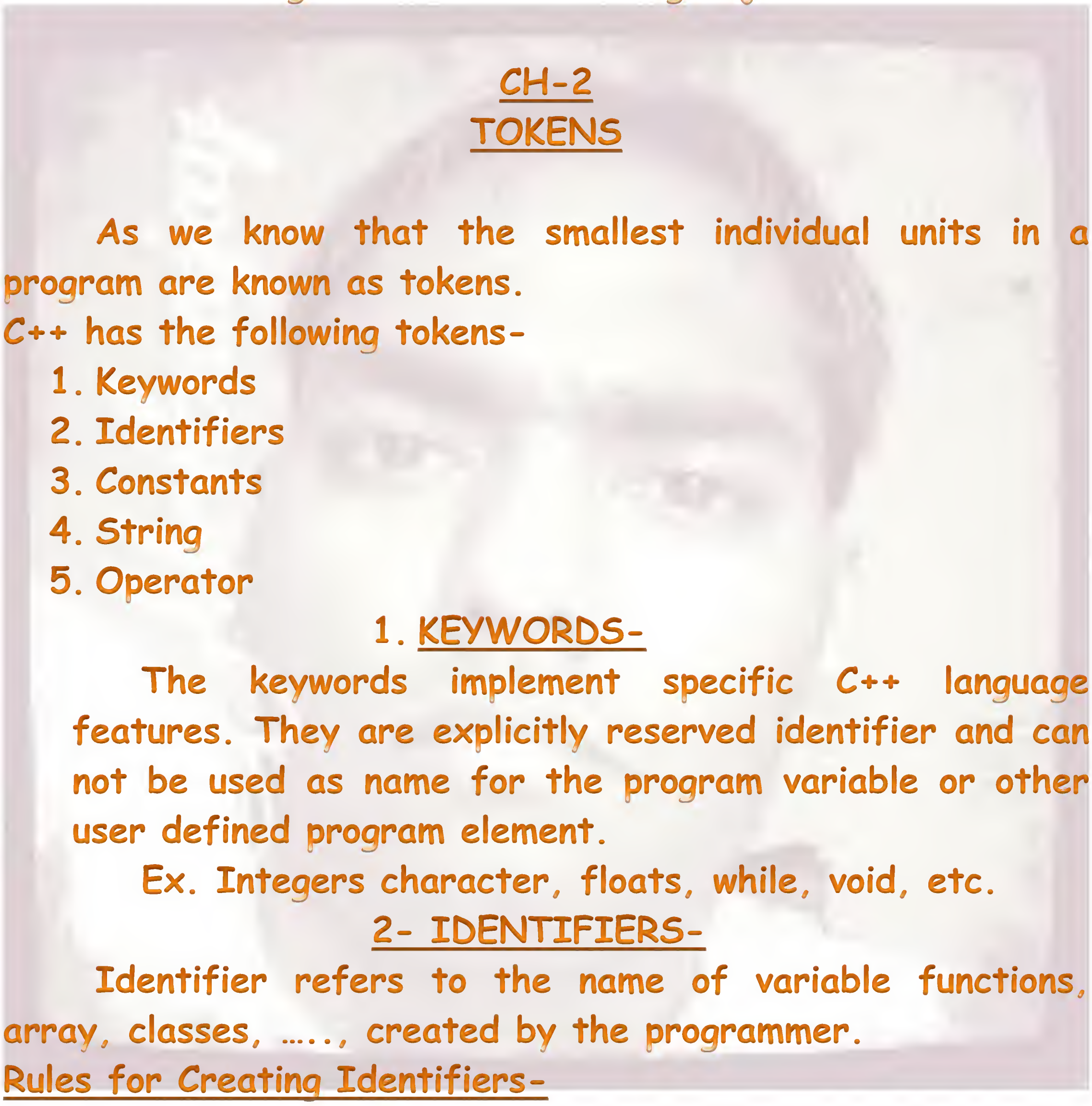
4- Polymorphism

Polymorphism is another important OOPS's concept it is a Greek word which means ability to take more than one form or multiple form from sing data.

5- Message Passing-

An object oriented program consists of a set of objects that communicate with each other. The process of programming in an object oriented language involves the following steps-

- Creating class that defines object and their behavior.
- Creating object form class definition.
- Establishing communication among objects.



CH-2
TOKENS

As we know that the smallest individual units in a program are known as tokens.

C++ has the following tokens-

1. Keywords
2. Identifiers
3. Constants
4. String
5. Operator

1. KEYWORDS-

The keywords implement specific C++ language features. They are explicitly reserved identifier and can not be used as name for the program variable or other user defined program element.

Ex. Integers character, floats, while, void, etc.

2- IDENTIFIERS-

Identifier refers to the name of variable functions, array, classes, ….., created by the programmer.

Rules for Creating Identifiers-

1- Only alphabetic characters, digits and under score are permitted.

2- The name can not be started with digit.

3- Upper case and lower case characters or letters are distinct because of case sensitive.

4- The declared keyword can not be used as variable.

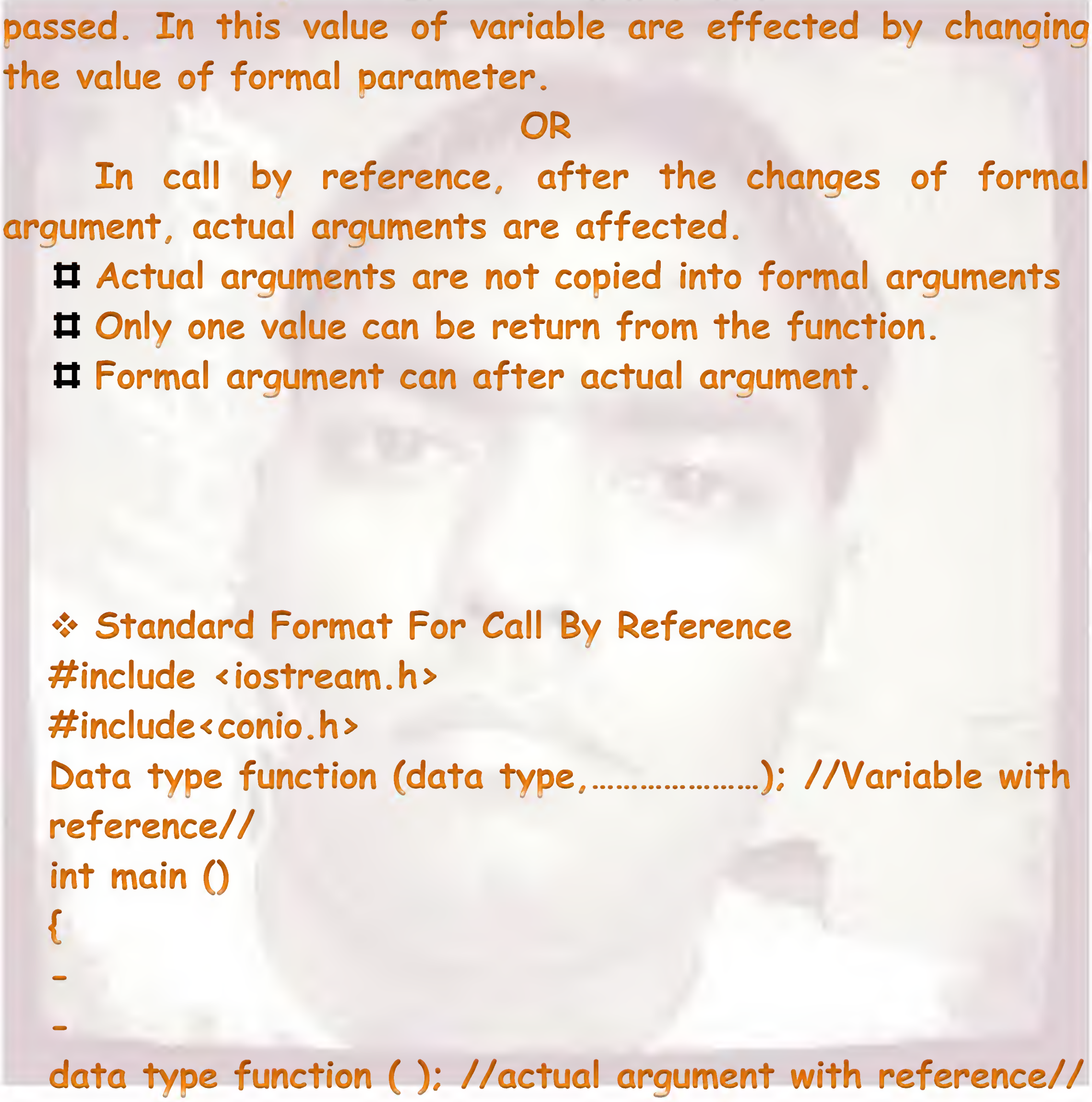
Passing Argument to the Function

There are two ways to pass the arguments in the function-

1-Call by reference (call by pointer)

2-Call by value

1- CALL BY REFERENCE-



In call by reference, the address of variable are passed. In this value of variable are effected by changing the value of formal parameter.

OR

In call by reference, after the changes of formal argument, actual arguments are affected.

⌗ Actual arguments are not copied into formal arguments

⌗ Only one value can be return from the function.

⌗ Formal argument can after actual argument.

❖ Standard Format For Call By Reference

```
#include <iostream.h>
#include<conio.h>
Data type function (data type,…………………); //Variable with
reference//
int main ()
{
-
-
data type function ( ); //actual argument with reference//
-
-
}
data type function ( data type ,……………) //formal argument
                                           data type with the pointer
                                           variable//
```

```
{
-
-
}
```

⇒ NOTE-

- ✓ Make the actual argument with reference (&).
- ✓ Make the formal argument with pointer (*).

2- CALL BY VALUE-

In call by value, the values of variable are passed. In this value of variable are not effected by changing in the value of formal parameter.

OR

In call by value, after the changing in the formal argument, actual argument is not effected.

- ⌗ Actual argument are copy into formal argument
- ⌗ Only one value can be return from the function
- ⌗ Formal argument can not be after actual argument
- ⌗ Formal argument are created after the passing the actual argument.

❖ Standard Format For Call By Value.

```
#include <iostream.h>
#include<conio.h>
data type function (data type,…………………);
int main ()
{
-
data type function ( ); //actual argument//
-
}
data type function ( data type ,……………) //formal argument
                                          with data type//
```

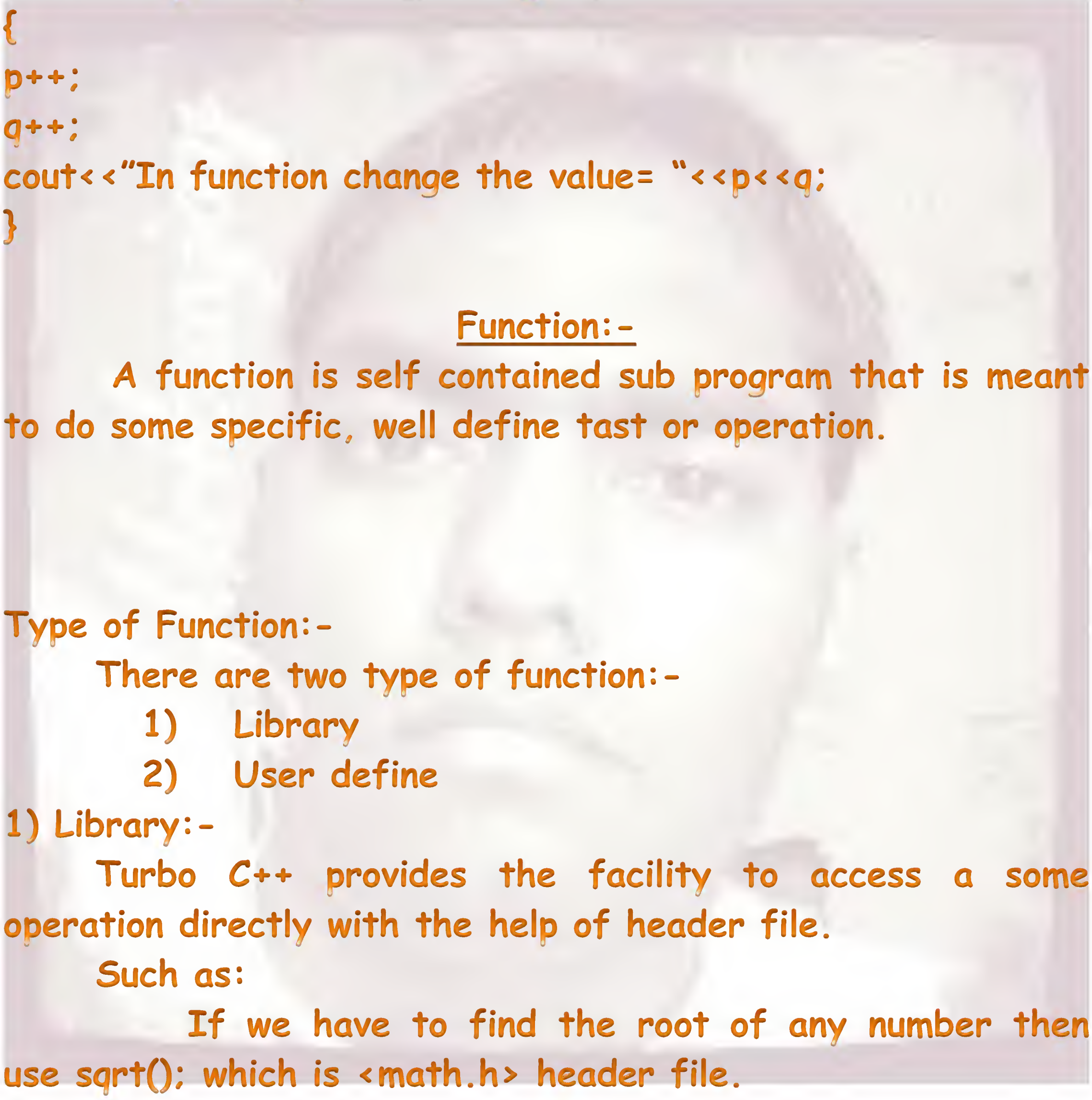
```
    {
    -
    -
    }
```

❖ Example For Call By reference.

```
#include<iostream.h>
#include<conio.h>
int Hi (int &a, int &b);
int main()
{
int a,b;
clrscr();
cout<<"Enter the value of a and b: ";
cin>>a>>b;
cout<<"Before calling the function a and b are=  "<<a<<b;
int Hi(&a,&b);
cout<<"After calling the function a and b are = "<<a<<b;
getch();
}
int Hi(int *p, int *q)
{
*p++;
*q++;
cout<<"In function change the value= "<<*p<<*q;
}
```

Example for call by value :-

```
#include<iostream.h>
#include<conio.h>
int Hi (int a, int b);
int main()
{
int a,b;
clrscr( );
cout<<"Enter the value of a and b: ";
cin>>a>>b;
cout<<"Before calling the function a and b are=  "<<a<<b;
int Hi(a,b);//actual argument//
cout<<"After calling the function a and b are = "<<a<<b;
getch( );
}
int Hi(int p, int q)//formal argument//
{
p++;
q++;
cout<<"In function change the value= "<<p<<q;
}
```



Function:-

A function is self contained sub program that is meant to do some specific, well define tast or operation.

Type of Function:-

There are two type of function:-

1) Library
2) User define

1) Library:-

Turbo C++ provides the facility to access a some operation directly with the help of header file.

Such as:

If we have to find the root of any number then use sqrt(); which is <math.h> header file.

```
<msyh.h>                   poe(x,2);
<string.h>        strlen( );
                  strcyp( );
                  strcat( );
                  strcmp( ) etc.
```

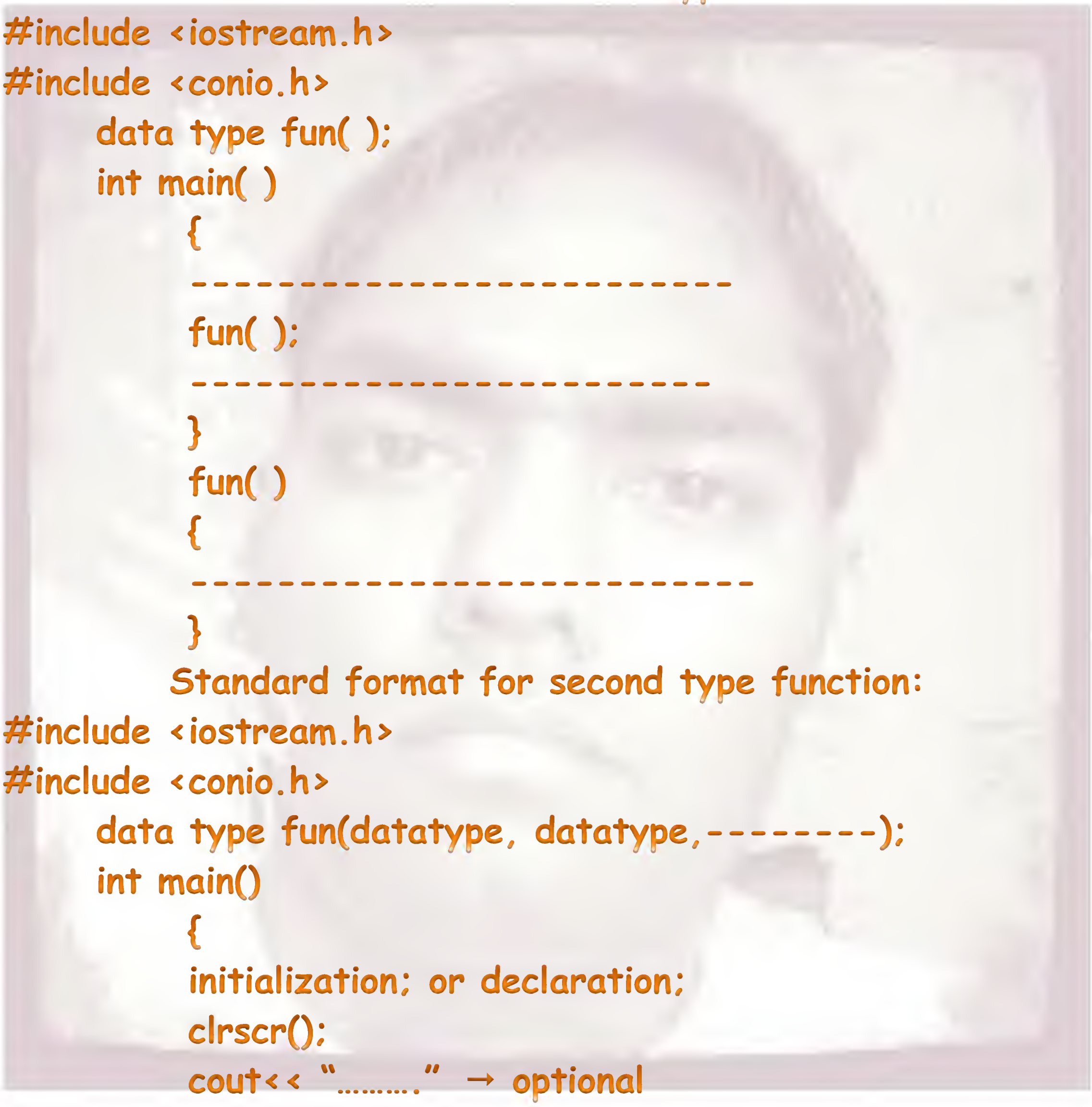
2)User define Function:-

The function which is created by user is called user define function.

Type of user define function:-

1) Function with no argument and no return value.
2) Function with argument but no return value.
3) Function with argument and return value.

Standard format for first type function:

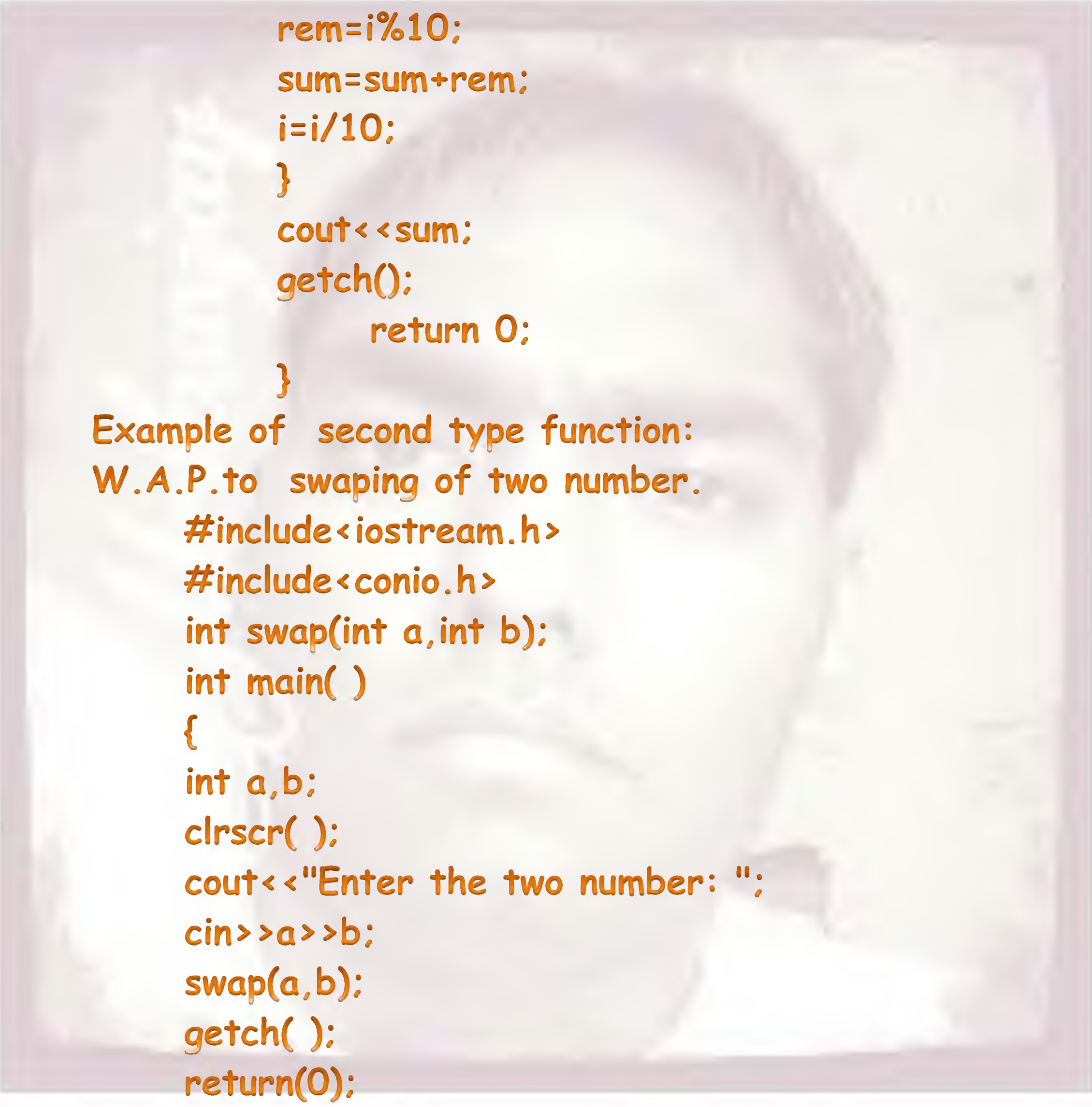


```
#include <iostream.h>
#include <conio.h>
     data type fun( );
     int main( )
          {
          -------------------------
          fun( );
          ------------------------
          }
          fun( )
          {
          --------------------------
          }
```

Standard format for second type function:

```
#include <iostream.h>
#include <conio.h>
     data type fun(datatype, datatype,--------);
     int main()
          {
          initialization; or declaration;
          clrscr();
          cout<< "………." → optional
          cin>>……;       →optional
          fun(datatype, datatype,--------);
          getch( );
          }
          fun(datatype, datatype,--------)
          {
            initialization;
            --------------
            }
            Standard format for third type function:
#include <iostream.h>
#include <conio.h>
      data type fun(datatype, datatype,--------);
      int main()
            {
            initialization;
            clrscr();
            cout<< "……….";          → optional
            cin>>……;                →optional
            fun(data type, data type,--------);
            cout<<fun(variable);
            getch ( );
            }
            fun (data type, data type,-----)
            {
            initialization;
            --------------
            return (result);
            }

      Exmple  of  first type function:
      W.A.P.to find the sum of digit.
      #include<iostream.h>
      #include<conio.h>
            SOD( );
     I            nt main( )
            {
            SOD( );
             return 0;
            }
             SOD( )
```



```
{
        int i,rem,sum=0;
        clrscr();
        cout<<"Enter the number: ";
        cin>>i;
         while(i>0)
        {
        rem=i%10;
        sum=sum+rem;
        i=i/10;
        }
        cout<<sum;
        getch();
                return 0;
        }
```

Example of second type function:

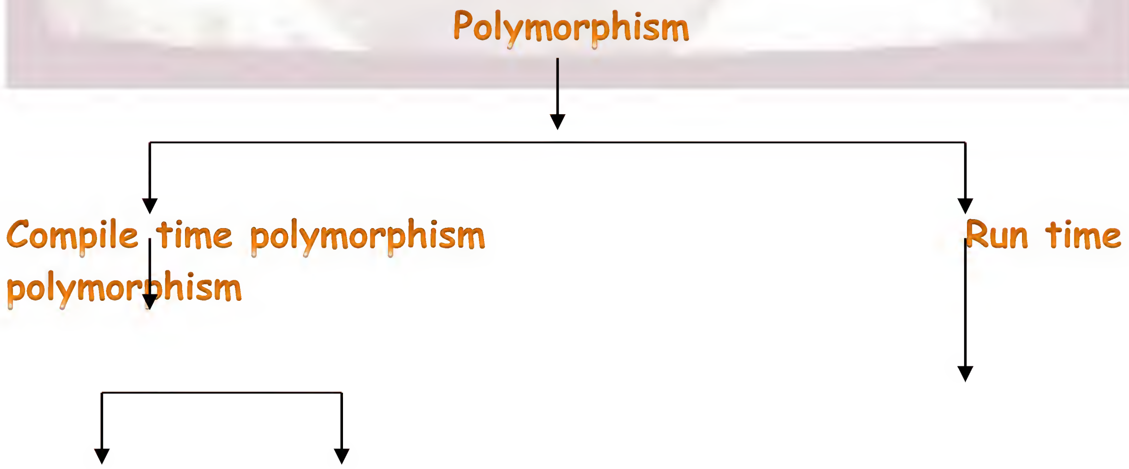
W.A.P.to swaping of two number.

```
#include<iostream.h>
#include<conio.h>
int swap(int a,int b);
int main( )
{
int a,b;
clrscr( );
cout<<"Enter the two number: ";
cin>>a>>b;
swap(a,b);
getch( );
return(0);
}
swap(int a,int b)
{
a=a+b;
b=a-b;
a=a-b;
cout<<a<<b;
return(0);
}
```

Example of thired type function:

W.A.P.to find the simple interest.

```
#include<iostream.h>
#include<conio.h>
 int SI(float p,float r,float t);
int main( )
{
float p,r,t;
clrscr( );
cout<<"Enter the value of p,r & t:";
cin>>p>>r>>t;
SI(p,r,t);
cout<<SI(p,r,t);
getch( );
return 0;
}
SI(float a,float b,float c)
{
float si;
si=(a*b*c)/100;
return (si);
}
```

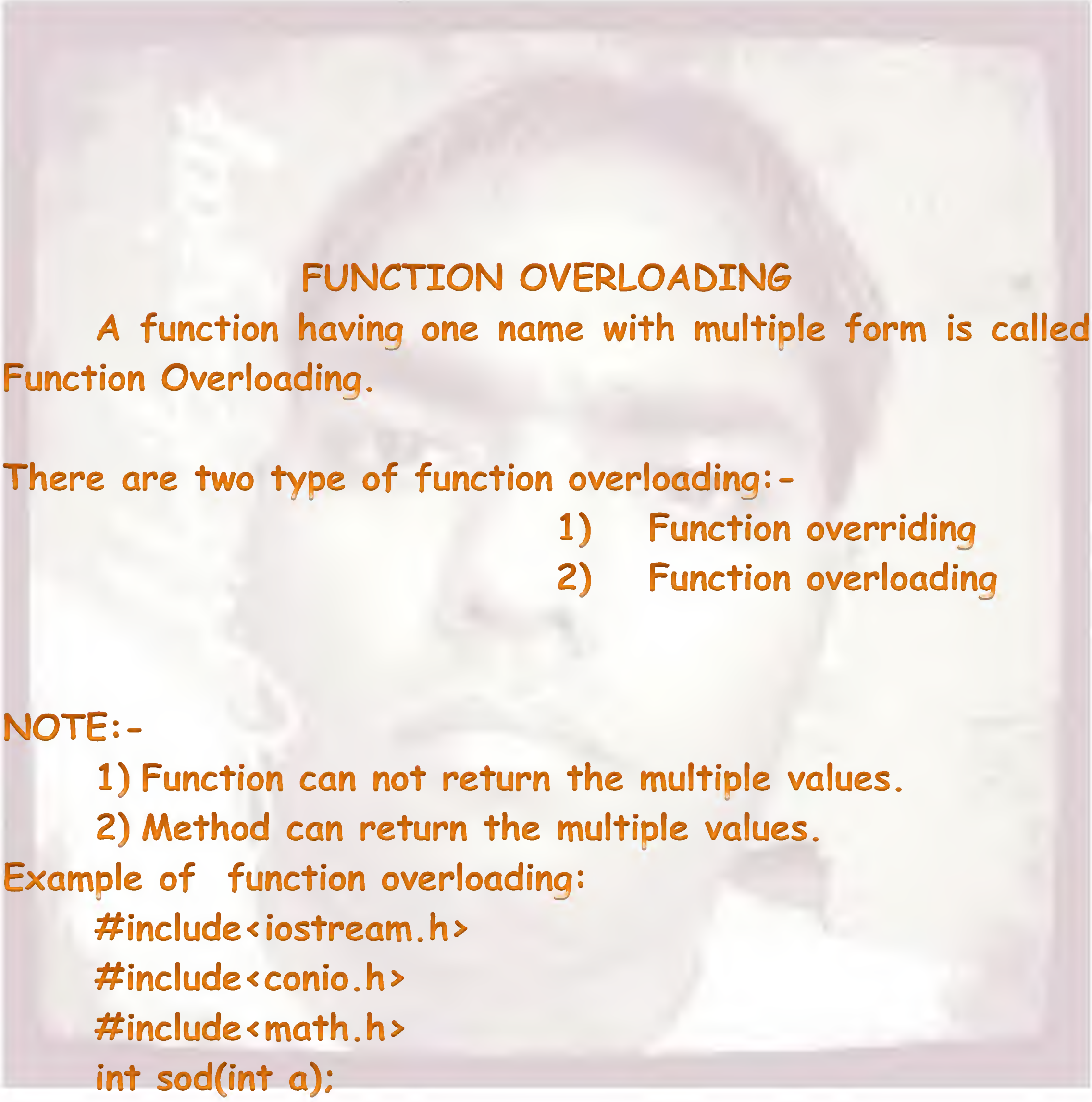


Polymorphism

Compile time polymorphism

Run time polymorphism

Function Operator
Virtual
overloading overloading
function



FUNCTION OVERLOADING

A function having one name with multiple form is called Function Overloading.

There are two type of function overloading:-

1) Function overriding
2) Function overloading

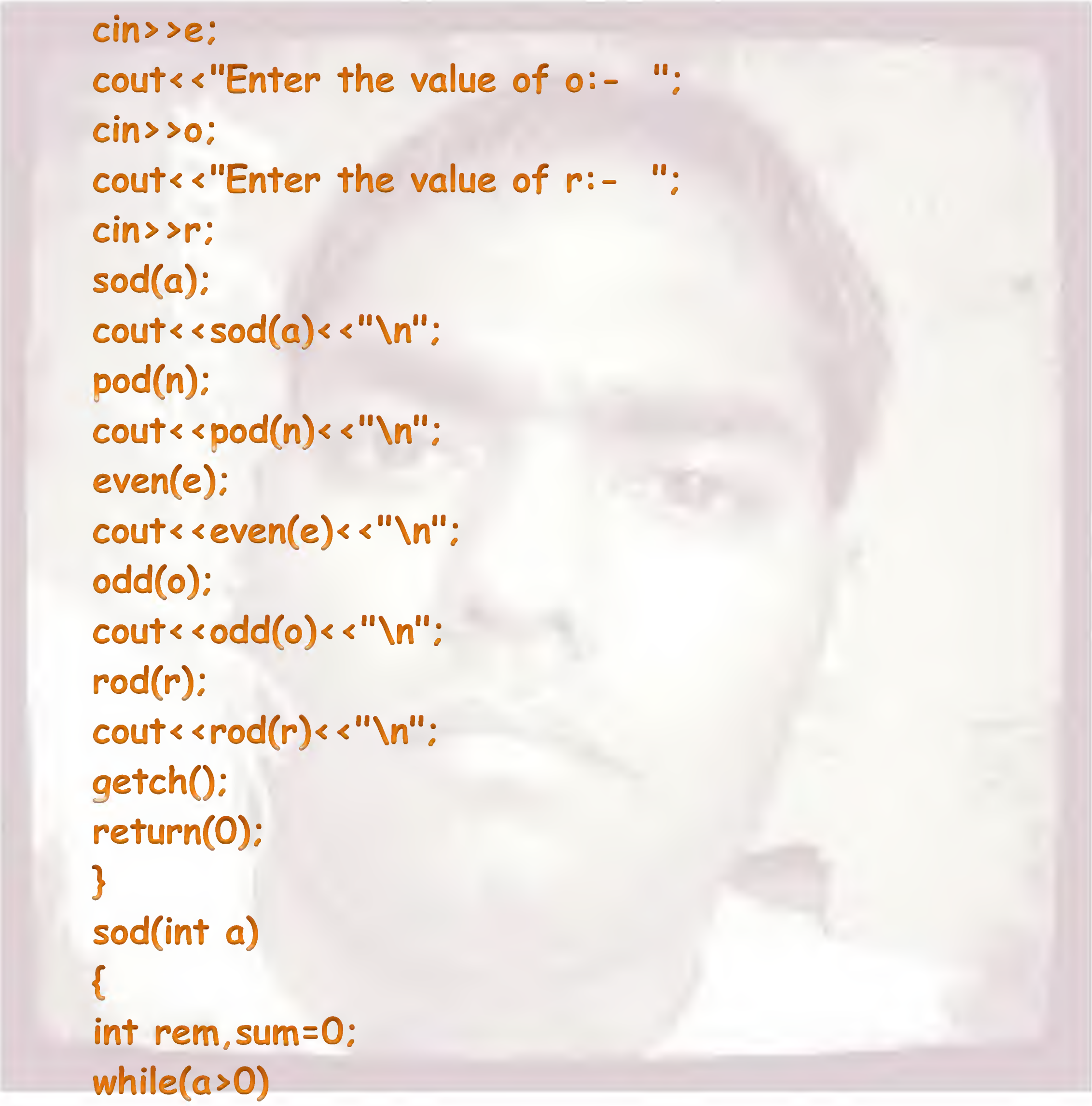
NOTE:-

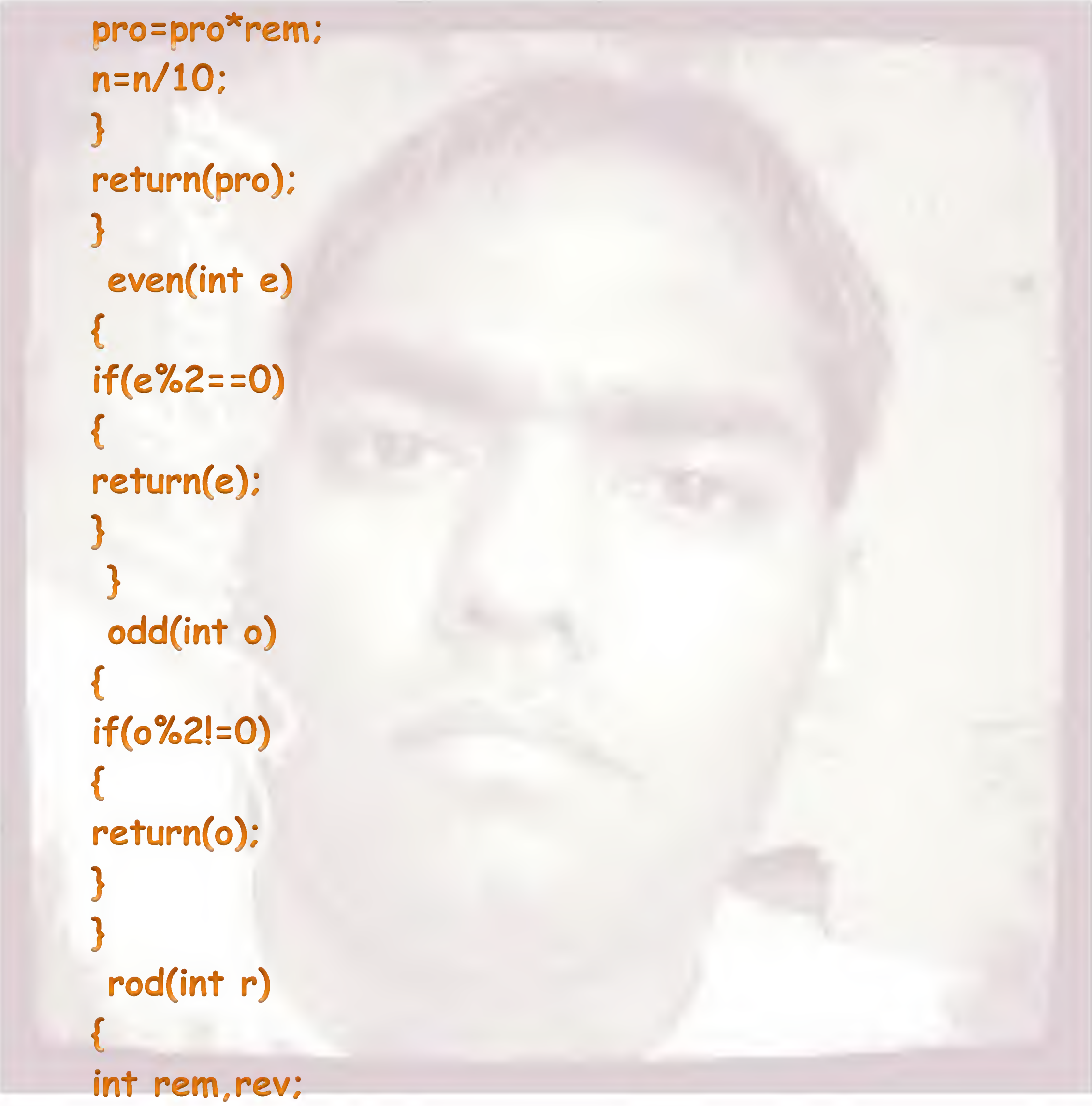
1) Function can not return the multiple values.
2) Method can return the multiple values.

Example of function overloading:

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
int sod(int a);
int pod(int n);
int even(int e);
int odd(int o);
int rod(int r);
int main()
{
```

```
int a,n,e,o,r;
clrscr();
cout<<"Enter the value of a,for the sum of digit :-   ";
cin>>a;
cout<<"Enter the value n:- ";
cin>>n;
cout<<"Enter the value of e:-  ";
cin>>e;
cout<<"Enter the value of o:-  ";
cin>>o;
cout<<"Enter the value of r:-  ";
cin>>r;
sod(a);
cout<<sod(a)<<"\n";
pod(n);
cout<<pod(n)<<"\n";
even(e);
cout<<even(e)<<"\n";
odd(o);
cout<<odd(o)<<"\n";
rod(r);
cout<<rod(r)<<"\n";
getch();
return(0);
}
sod(int a)
{
int rem,sum=0;
while(a>0)
{
rem=a%10;
sum=sum+rem;
a=a/10;
}
return(sum);
```





```
}
 pod(int n)
{
int rem,pro;
while(n>0)
{
rem=n%10;
pro=pro*rem;
n=n/10;
}
return(pro);
}
 even(int e)
{
if(e%2==0)
{
return(e);
}
 }
 odd(int o)
{
if(o%2!=0)
{
return(o);
}
}
 rod(int r)
{
int rem,rev;
while(r>0)
{
rem=r%10;
rev=rev+rem*rem*rem;
r=r/10;
}
```

```
return(rev);
}
```

INLINE FUNCTION

To eleminet the cost of the call to small function C++ provide a new function called Inline Function.

An Inline Function is function that is expended inline when its calls.

Key word of Inline Function → inline.

Standard format for Inline function:

```
#include <iostream.h>
#include <conio.h>
      inline data type fun(data type & argument)
            {
                  body of function
            }
      int main ( )
            {
            Initialization or declaration;
            data type fun(); //with actual argument//
            cout<<fun();    //with actual argument//
            }
```

LIMITATION OF INLINE FUNCTION

Some of the situation where inline function not work.

1) For function returning value, if a loop, a switch, or go to statement exit.
2) For function can not return value if return statement is exit.
3) If function contain static variable.
4) If Inline Function recursive function.

NOTE:-

Inline Function made a program run faster because the overload of a function call and return is eliminate.

Example of inline function :

```
#include<iostream.h>
#include<conio.h>
inline float mult(float a,float b)//use inline function//
{
return (a*b);
}
inline float div(float a,float b) //use inline function//
{
return (a/b);
}
int main()
{
float a,b;
clrscr( );
cout<<"Enter the two number : ";
cin>>a>>b;
mult(a,b);
cout<<mult(a,b)<<"\n";
div(a,b);
cout<<div(a,b);
getch( );
return(0);
}
```

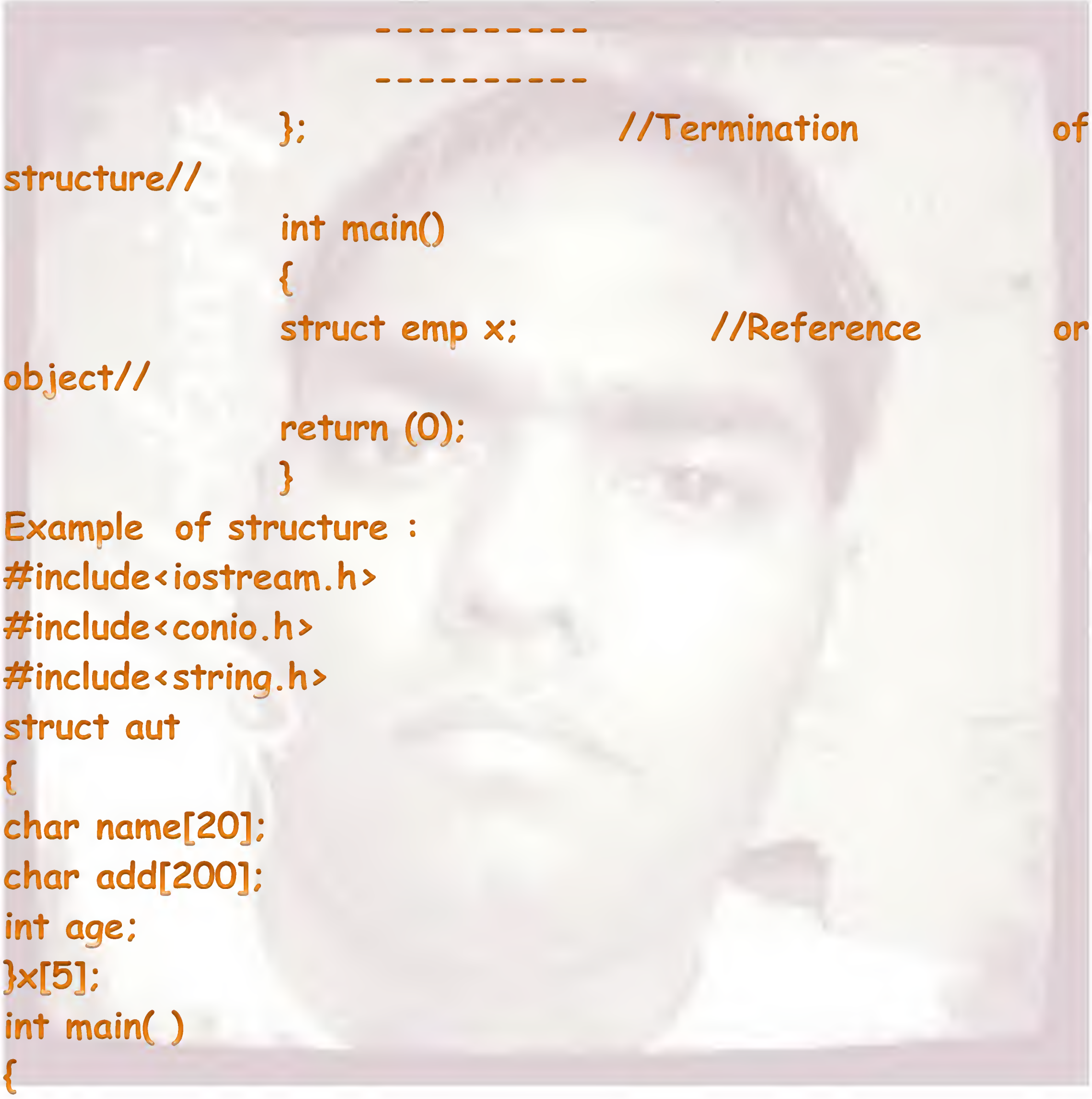
STRUCTURE

As we known that array collection of similar type of data element but it may be possible to take different type of element together. It is useful to group different type of data which has some meaningful information using.

An other words , collection of different type data element is called structure.

Struct is keyword of structure.

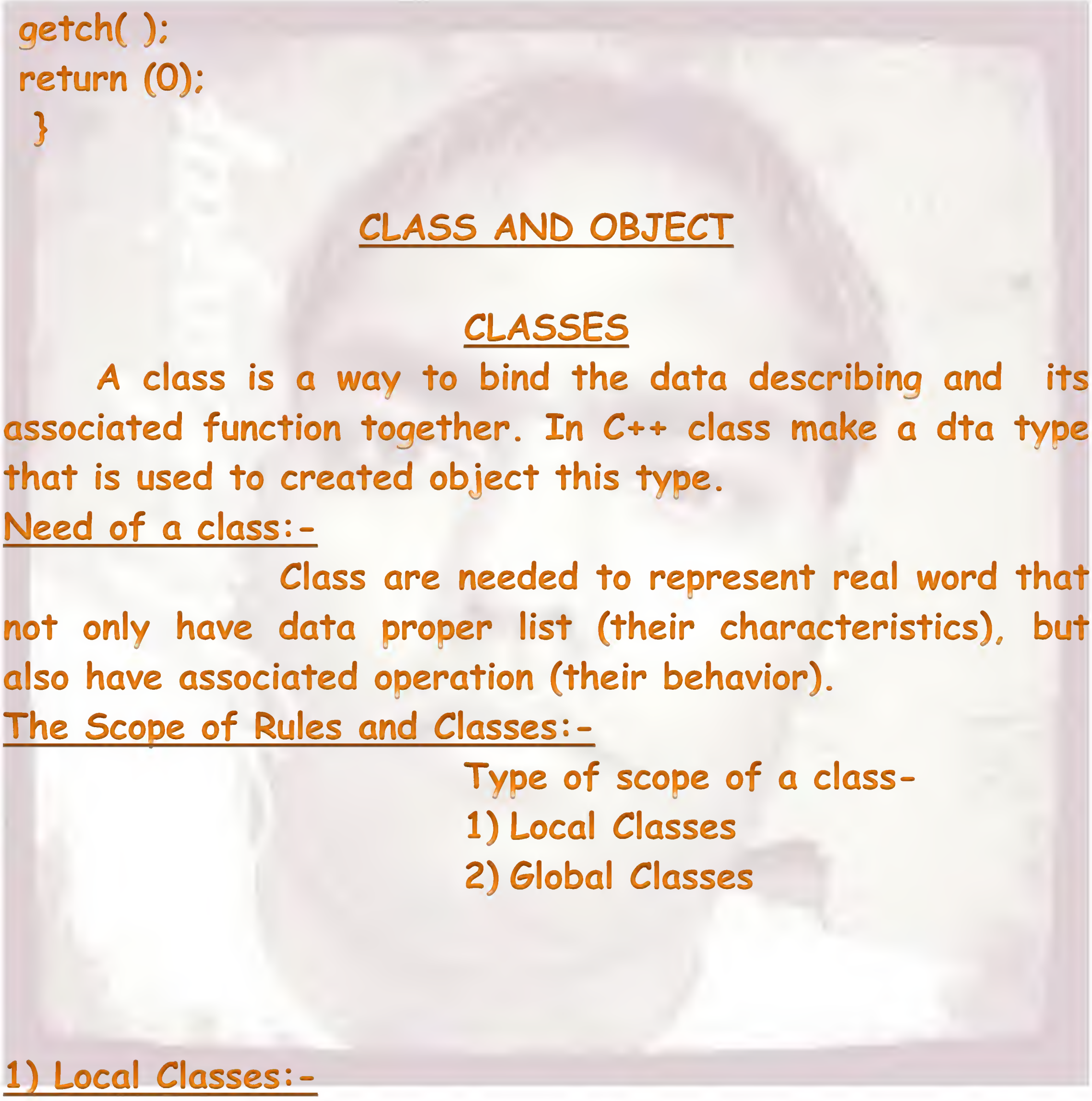
Standard format for structure.



```
Keyword of structure ←  struct emp; → Name of structure
                        {
                              ----------
                              ----------
                        };                //Termination of structure//
                        int main()
                        {
                        struct emp x;           //Reference or object//
                        return (0);
                        }
```

Example of structure :

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
struct aut
{
char name[20];
char add[200];
int age;
}x[5];
int main( )
{
int i;
clrscr ( );
for(i=0;i<5;i++)
{
cout<<"Enter the name,age &address\n: ";
cin>>x[i].name>>x[i].age>>x[i].add;
}
for(i=0;i<5;i++)
{
cout<<"\nAuther Name="<<x[i].name;
cout<<"\nAge="<<x[i].age;
cout<<"\nAddress="<<x[i].add;
 }
```



```
 getch( );
 return (0);
  }
```

CLASS AND OBJECT

CLASSES

A class is a way to bind the data describing and its associated function together. In C++ class make a dta type that is used to created object this type.

Need of a class:-

Class are needed to represent real word that not only have data proper list (their characteristics), but also have associated operation (their behavior).

The Scope of Rules and Classes:-

Type of scope of a class-

1) Local Classes

2) Global Classes

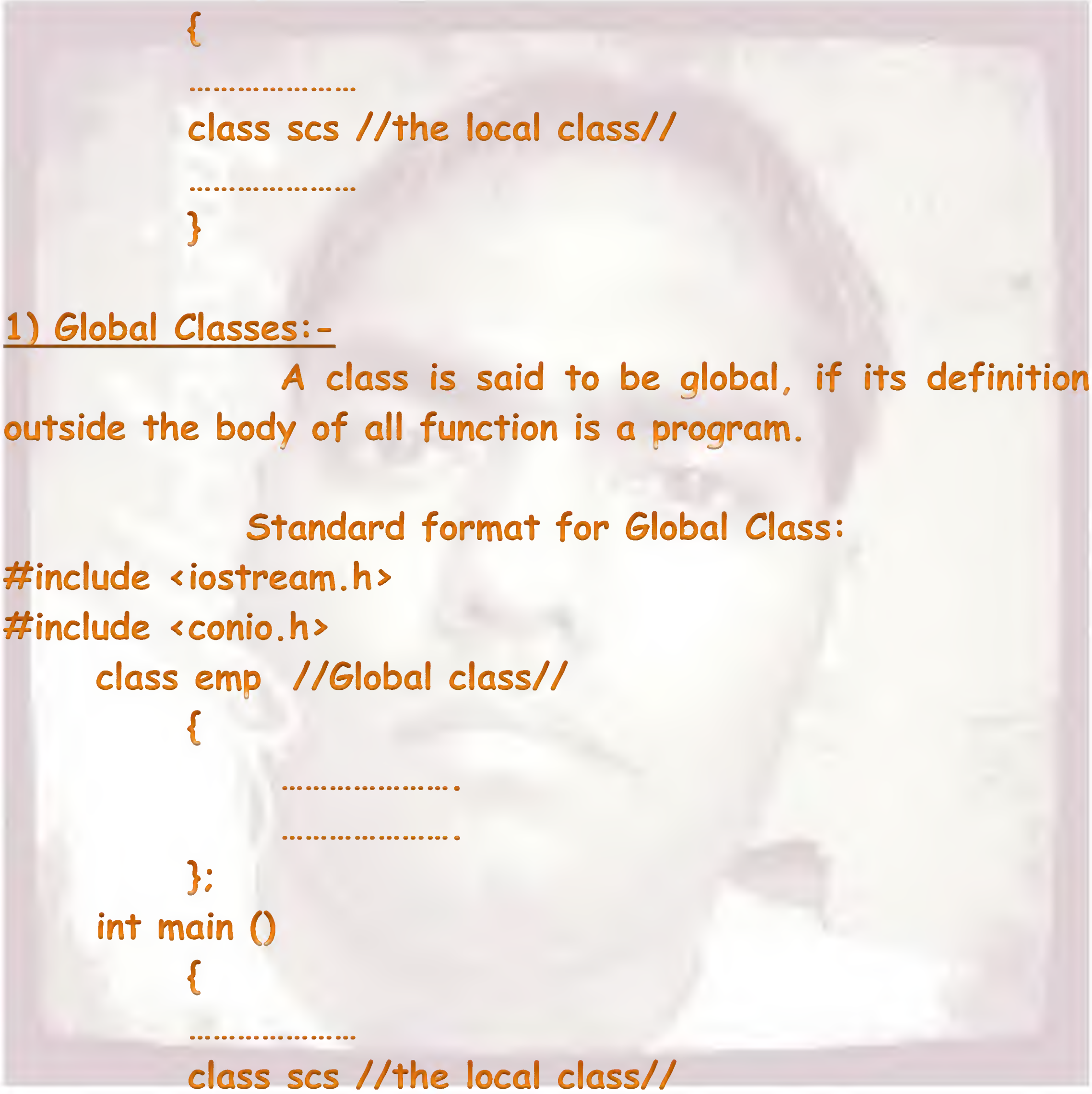
1) Local Classes:-

A class said to be local if its definition occur in side a function body, which means that object of this classes type can be declared only within the function that defined this class type.

Standard format for Local Class:

```
#include <iostream.h>
```

```
#include <conio.h>
        class emp
                {
                        ………………….
                        ………………….
                };
        int main ()
                {
                ………………….
                class scs //the local class//
                ………………….
                }
```



1) Global Classes:-

A class is said to be global, if its definition outside the body of all function is a program.

Standard format for Global Class:

```
#include <iostream.h>
#include <conio.h>
        class emp  //Global class//
                {
                        ………………….
                        ………………….
                };
        int main ()
                {
                ………………….
                class scs //the local class//
                ………………….
                }
```

Difference between Structure (C) and Class (C++)

Structure (C)	Class (C++)
1) C does not having any classes, C has only structure.	1) C++ having both classes and structure.
2) It is not secure, open ended.	2) It has good security, close ended.

3) Structure has a single black of statement such as
struct emp
{
……………………..
};
	3) Class can be divided into two part such as
	class emp
	{
	……………… //data member//
	______________________
	……………….//member function//
	};

4) It has no any access identifier. By default it has public access identifier.	4) It has tree type of access identifier 1) Public (P.C.O.) 2) Private (cell phone) 3) Protected. (land line)

NOTE:-

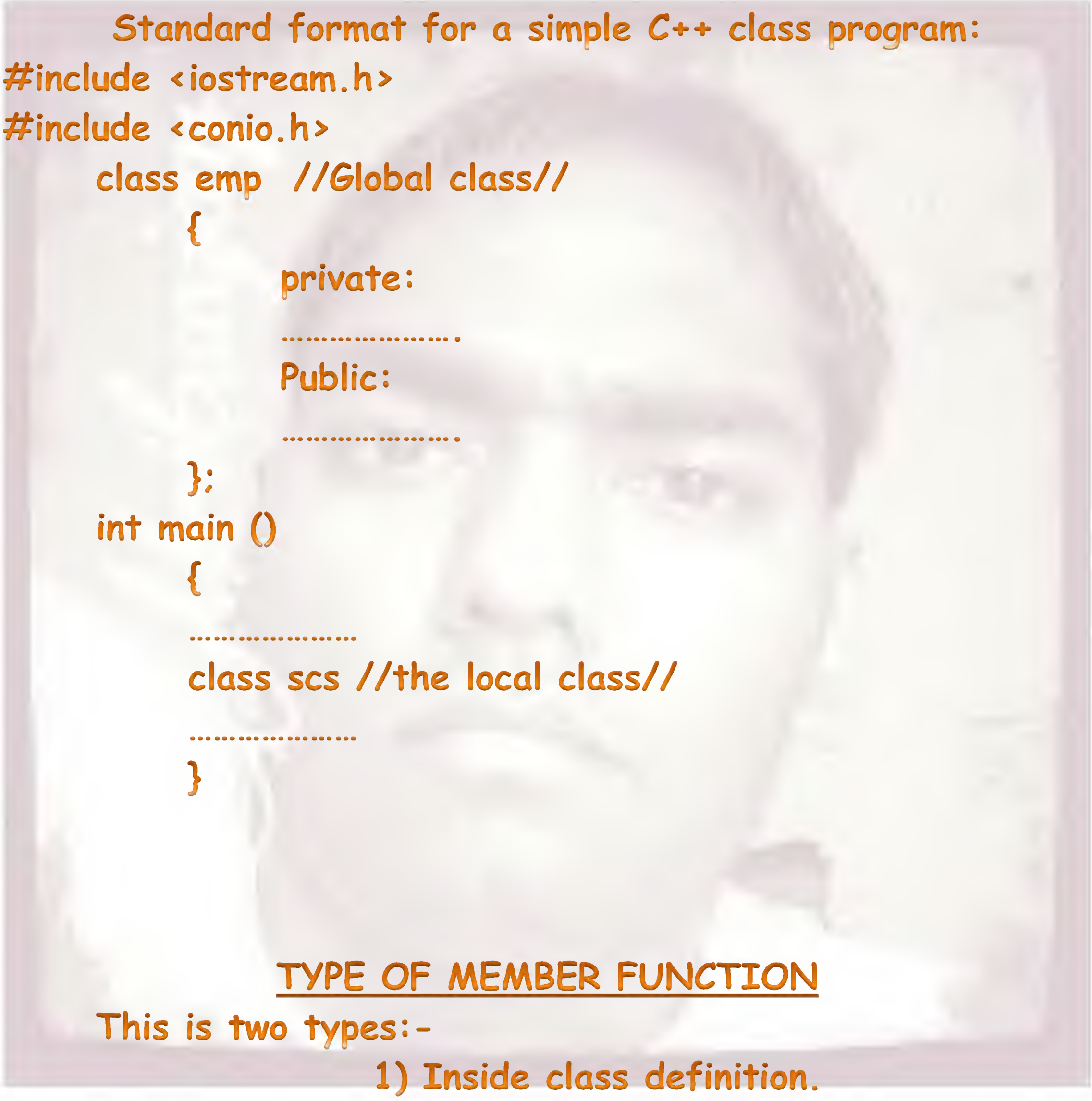
The only difference between structure and class in C++ is that by default a member of a class are private, while by default the member of structure are public.

MEMBER FUNCTION

It's a method by which we can access the private data, protected data.

DATA MEMBER

It is a part of class where we initialization type of the data, same as structure.



Standard format for a simple C++ class program:

```
#include <iostream.h>
#include <conio.h>
      class emp  //Global class//
            {
                  private:
                  ………………….
                  Public:
                  ………………….
            };
      int main ()
            {
            ………………….
            class scs //the local class//
            ………………….
            }
```

TYPE OF MEMBER FUNCTION

This is two types:-

1) Inside class definition.
2) Outside class definition.

1) INSIDE CLASS DEFINITION

In side class definition we have to defined the function inside the class or within the class.

OR

Member function defined inside the class is called inside member function .

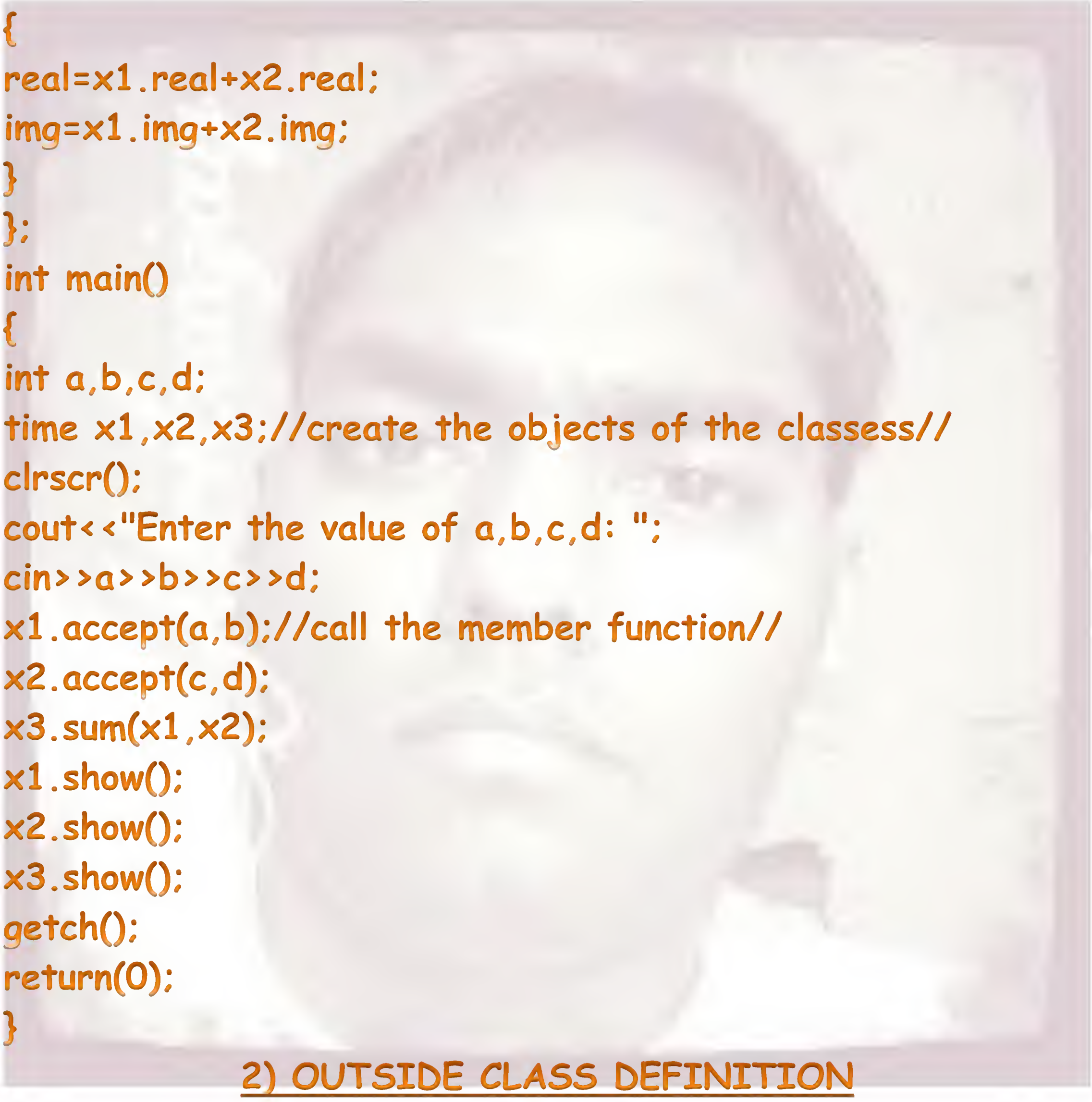
Standard format for inside class definition:

```
#include <iostream.h>
#include <conio.h>
     class emp  //Global class//
          {
               private:
               …………………….//data member//
               Public:
               void accept ()//inside definition of member
function//
               {
               ---------
               }
          };
     int main ()
          {
          …………………..
          class scs //the local class//
          …………………..
          }
```

Example of inside class definition:

WAP to find the sum of two imaginary number:

```
#include<iostream.h>
#include<conio.h>
class time
{
int real,img;
public:
void accept(int a,int b) //declare and definition  the member
function//
{
real=a;
img=b;
}
void show()//declare and definition of the member function//
{
cout<<real<<"+"<<img<<"i"<<"\n";
}
void sum(time x1,time x2) //declare and definition of  the
member function//
```



```
{
real=x1.real+x2.real;
img=x1.img+x2.img;
}
};
int main()
{
int a,b,c,d;
time x1,x2,x3;//create the objects of the classess//
clrscr();
cout<<"Enter the value of a,b,c,d: ";
cin>>a>>b>>c>>d;
x1.accept(a,b);//call the member function//
x2.accept(c,d);
x3.sum(x1,x2);
x1.show();
x2.show();
x3.show();
getch();
return(0);
}
```

2) OUTSIDE CLASS DEFINITION

In outside class definition we have to defined the function outside class.

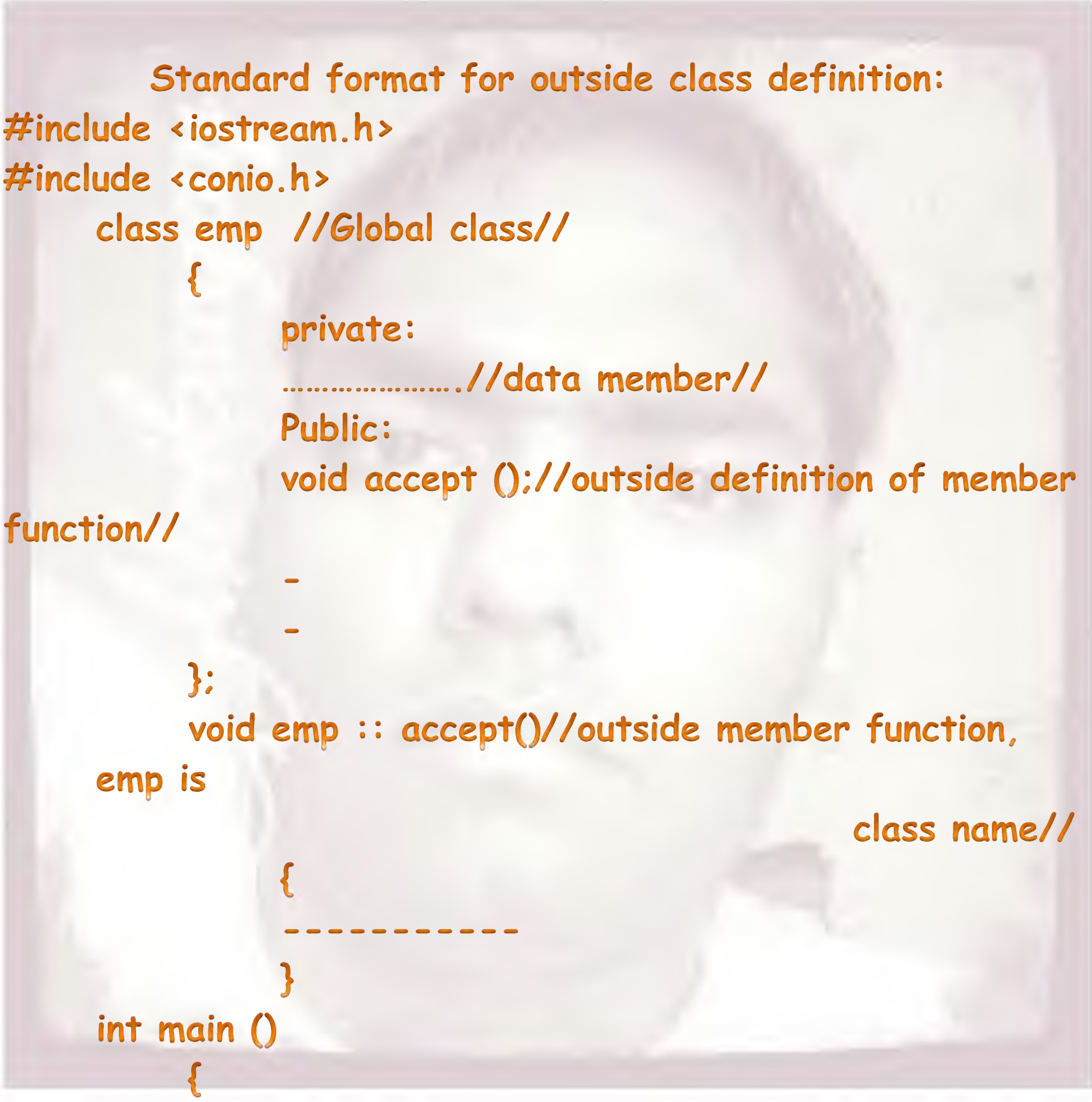
OR

A function which defined outside the class is called outside member function.

It is access by (::) scope resolution operator.

Scope resolution operator (::)

The symbol :: this is called scope resolution operator. Which is specify the scope of the function that to the class name. it is use to access the member function outside the class.

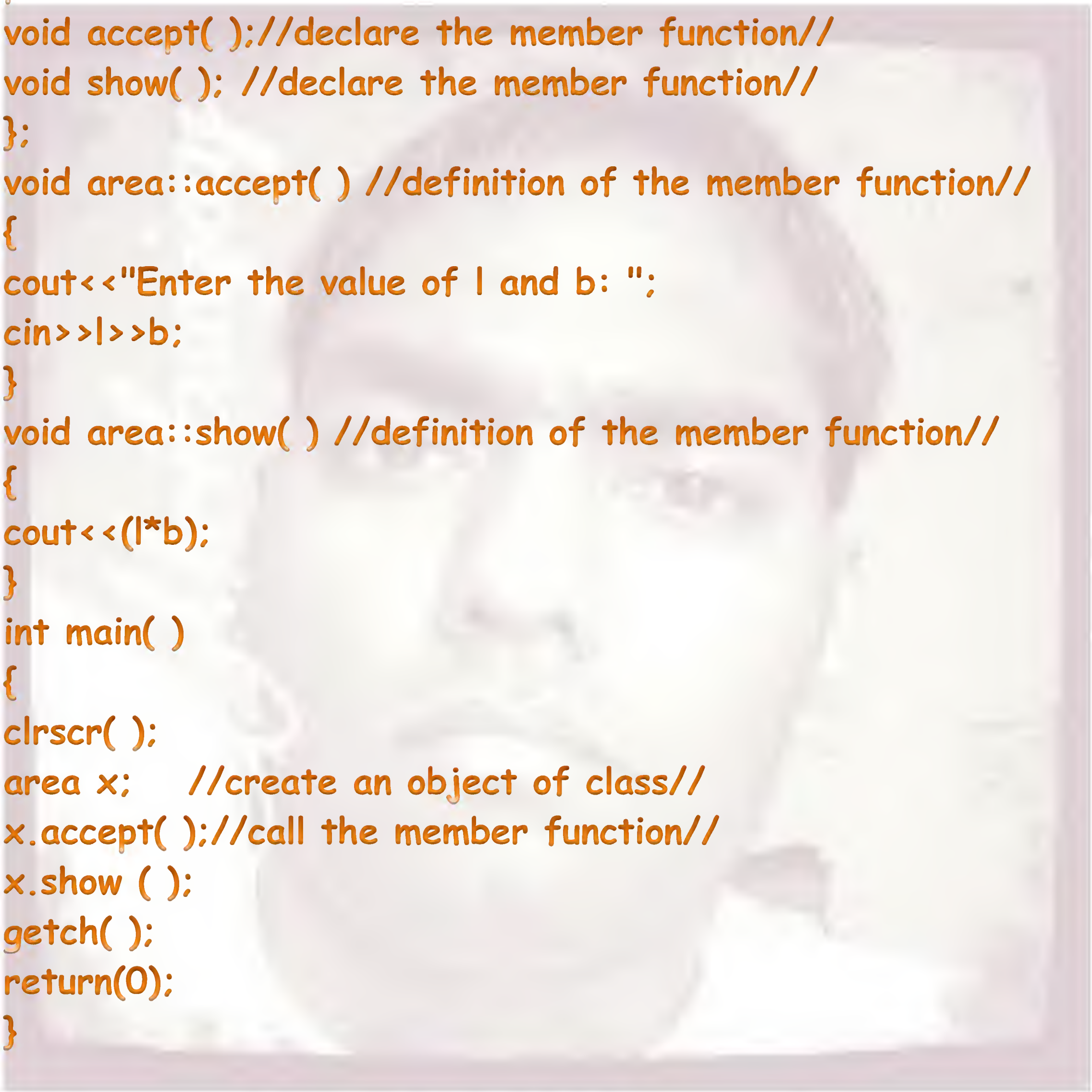


Standard format for outside class definition:

```
#include <iostream.h>
#include <conio.h>
      class emp  //Global class//
            {
                  private:
                  …………………….//data member//
                  Public:
                  void accept ();//outside definition of member
function//
                  -
                  -
            };
            void emp :: accept()//outside member function,
      emp is
                                          class name//
                  {
                  -----------
                  }
      int main ()
            {
            emp x;                              //x is object of
class//
            x.accept();             //call the member function//
            return(0);
            }
```

Example of outside class definition:

WAP to find the area of rectangle:



```
#include<iostream.h>
#include<conio.h>
class area
{
float l,b;
public:
void accept( );//declare the member function//
void show( ); //declare the member function//
};
void area::accept( ) //definition of the member function//
{
cout<<"Enter the value of l and b: ";
cin>>l>>b;
}
void area::show( ) //definition of the member function//
{
cout<<(l*b);
}
int main( )
{
clrscr( );
area x;     //create an object of class//
x.accept( );//call the member function//
x.show ( );
getch( );
return(0);
}
```

FRIEND FUNCTION

1) To access the private data of a class from without member function or non-member function.

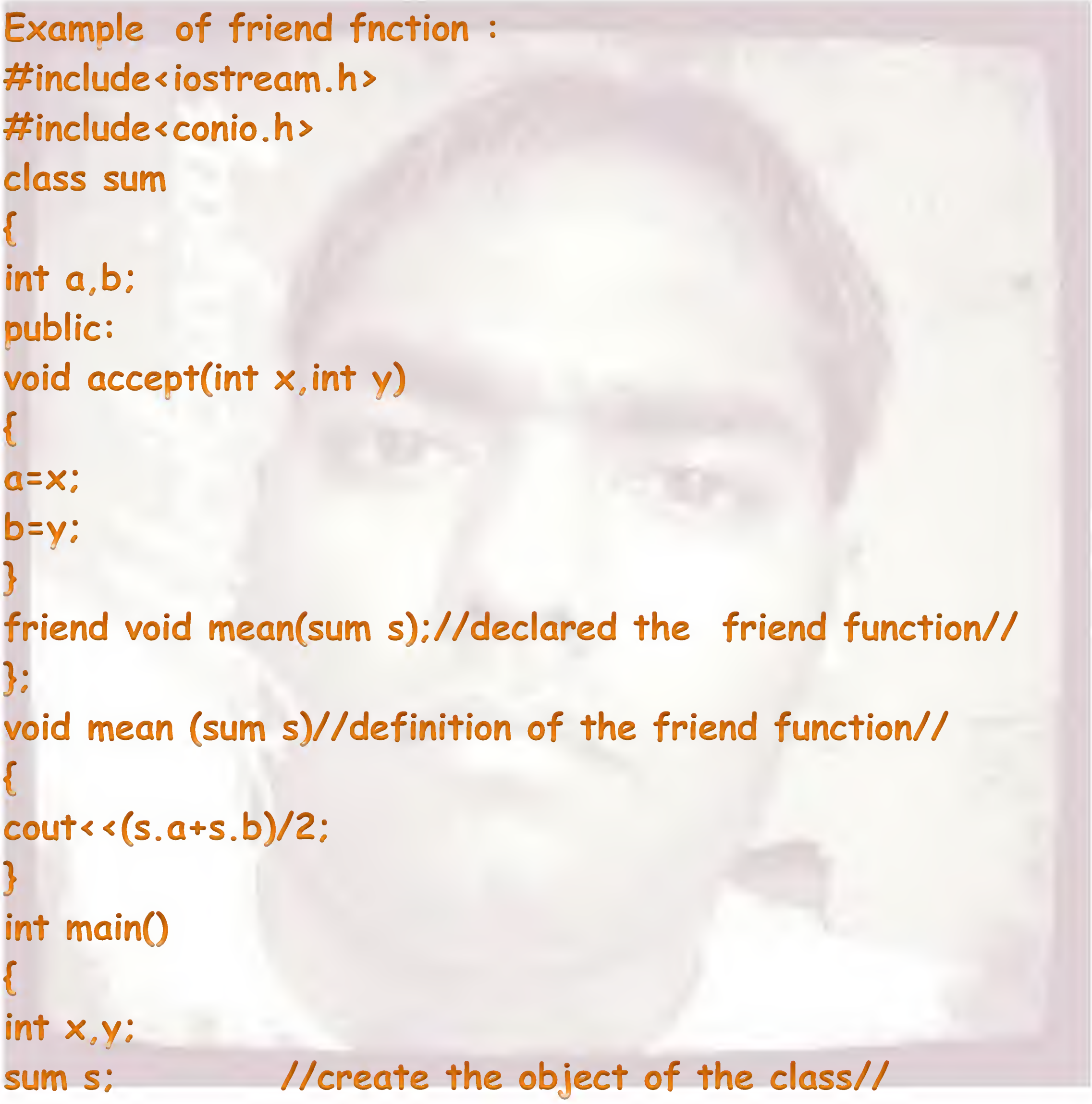
2) Increase the variability(efficiency) of a overload operator.

Characteristics of the friend function

1) It is not in the scope of the class, to which to has been declared. Since it is not in the scope of the class. It can not be called using the object of the class.
2) It can be invoked like normal function without help of any object.
3) It can be declared either in the public or the private part of the class without effecting its meaning.
4) Usually it has object as argument.
5) It is define outside the class without using scope resolution operator.

Standard format for Friend function:

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
     class emp
          {
          ------------------
          public:
          void accept ( )
          {
          ------------------
          }
          friend void display( );//using the friend function//
          };
          void display()
          {
          ------------------
}
int main()
{
emp x;
------------------
return (0);
}
```



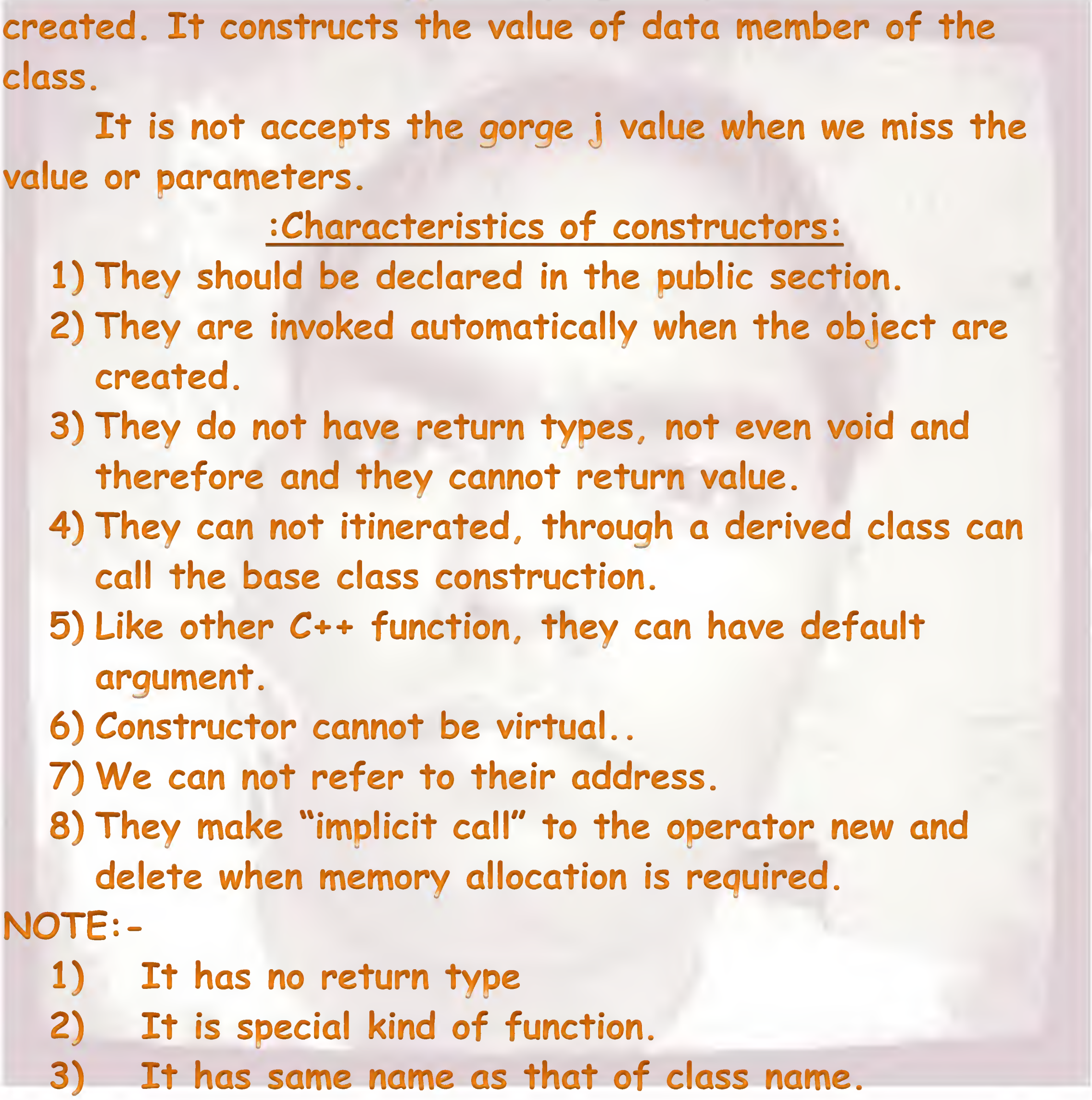
Example of friend fnction :

```
#include<iostream.h>
#include<conio.h>
class sum
{
int a,b;
public:
void accept(int x,int y)
{
a=x;
b=y;
}
friend void mean(sum s);//declared the  friend function//
};
void mean (sum s)//definition of the friend function//
{
cout<<(s.a+s.b)/2;
}
int main()
{
int x,y;
sum s;             //create the object of the class//
clrscr();
cout<<"Enter the two number: ";
cin>>x>>y;
s.accept(x,y);         //call the member function//
mean(s);
getch();
return(0);
}
```

CONSTRUCTORS AND DESTRUCTORS

:CONSTRUCTORS:

A constructor is a special member function because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is



created. It constructs the value of data member of the class.

It is not accepts the gorge j value when we miss the value or parameters.

:Characteristics of constructors:

1) They should be declared in the public section.
2) They are invoked automatically when the object are created.
3) They do not have return types, not even void and therefore and they cannot return value.
4) They can not itinerated, through a derived class can call the base class construction.
5) Like other C++ function, they can have default argument.
6) Constructor cannot be virtual..
7) We can not refer to their address.
8) They make "implicit call" to the operator new and delete when memory allocation is required.

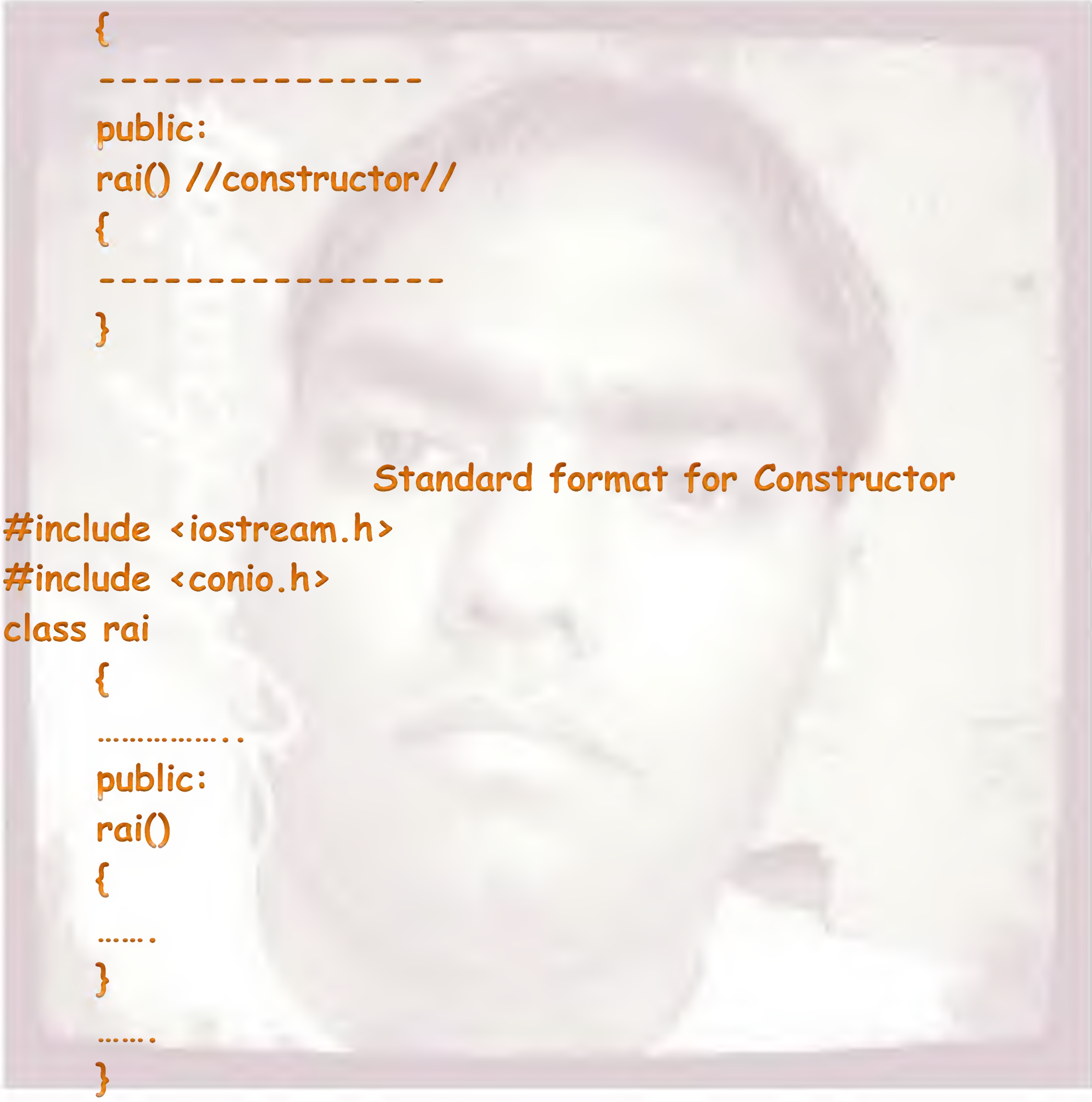
NOTE:-

1) It has no return type
2) It is special kind of function.
3) It has same name as that of class name.
4) If we not declared any constructor then
5) A program having any number of constructor but one constructor most wanted.
6) Default constructor can be created by user or compiler.

7) A constructor called automatically, when object is declared.

8) Pass the argument.

9) It is used for the crating memory allocation (dynamic memory allocation).



Ex.

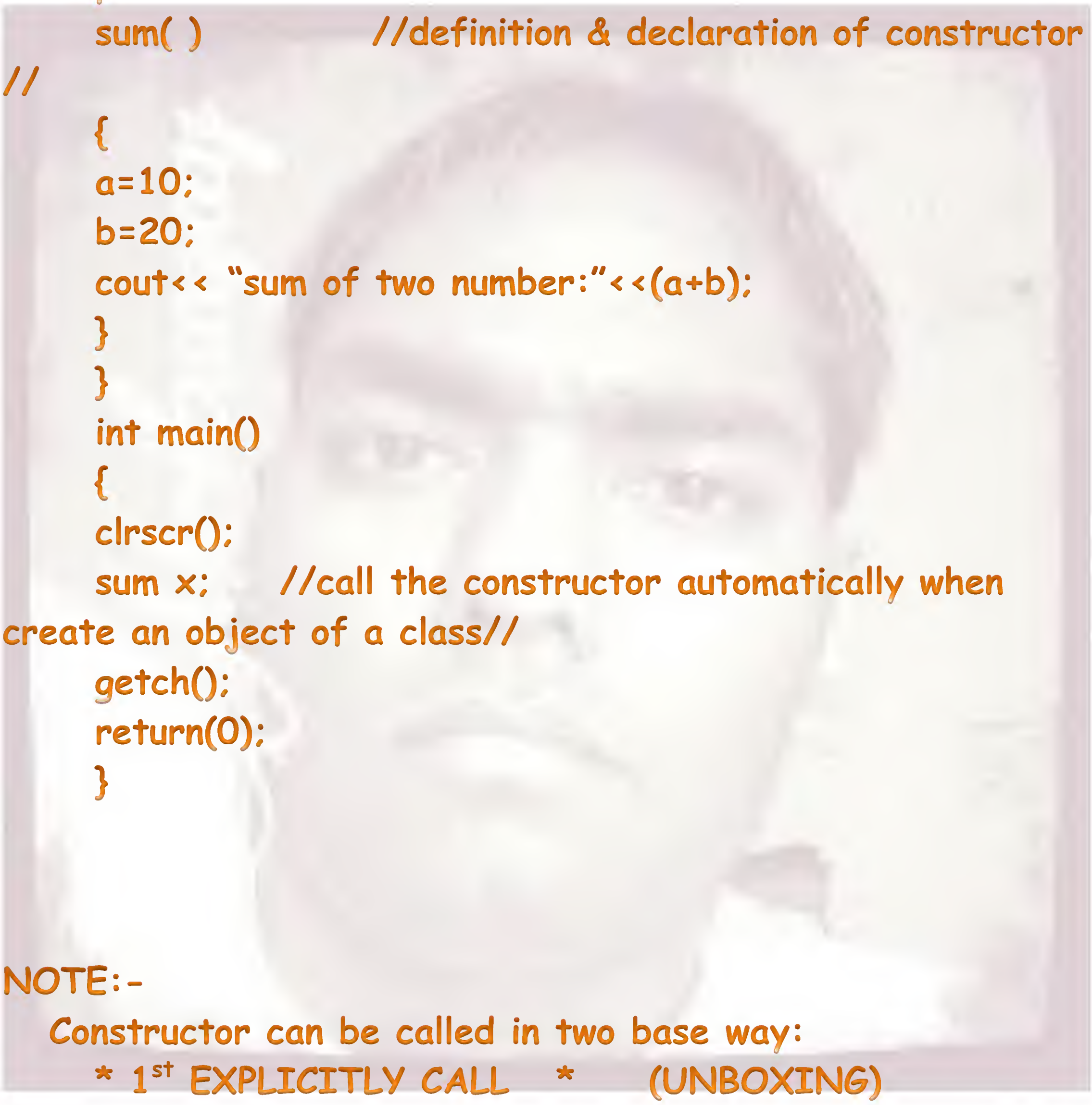
```
class rai //class name//
{
----------------
public:
rai() //constructor//
{
-----------------
}
```

Standard format for Constructor

```
#include <iostream.h>
#include <conio.h>
class rai
    {
    ……………..
    public:
    rai()
    {
    …….
    }
    …….
    }
int main ()
    {
    rai x;
    ……….
    return ();
    }
```

Example of Constructor:

```
#include <iostream.h>
#include <conio.h>
class sum
      {
      int a, b;
      public:
      sum( )              //definition & declaration of constructor
//
      {
      a=10;
      b=20;
      cout<< "sum of two number:"<<(a+b);
      }
      }
      int main()
      {
      clrscr();
      sum x;      //call the constructor automatically when
create an object of a class//
      getch();
      return(0);
      }
```



NOTE:-

Constructor can be called in two base way:

* 1st EXPLICITLY CALL * (UNBOXING)

(Pass the argument any how) (float →int)

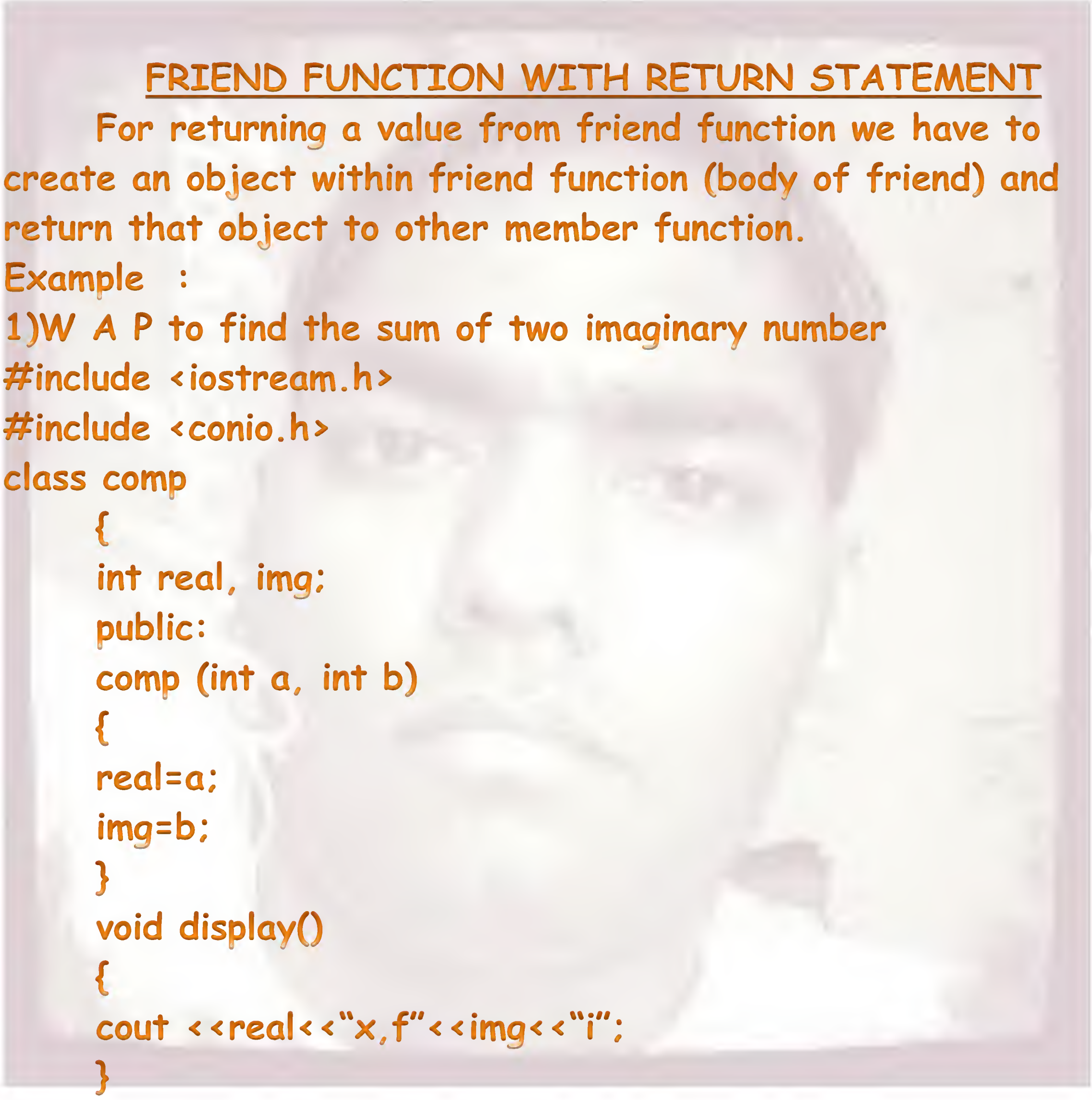
emp x = emp (10, 20);

Class name Object Class Name Argument

*2nd IMPLICITLY CALL * (BOXING)

(Match the argument properly) (int →float)

```
        emp          x              (10, 20);
        Class name   Object             Argument
        *NORMAL*
                exp          x;
object                    ← x,   accept             (10, 20);
        dd operator      Member    Argument
```



FRIEND FUNCTION WITH RETURN STATEMENT

For returning a value from friend function we have to create an object within friend function (body of friend) and return that object to other member function.

Example :

1)W A P to find the sum of two imaginary number

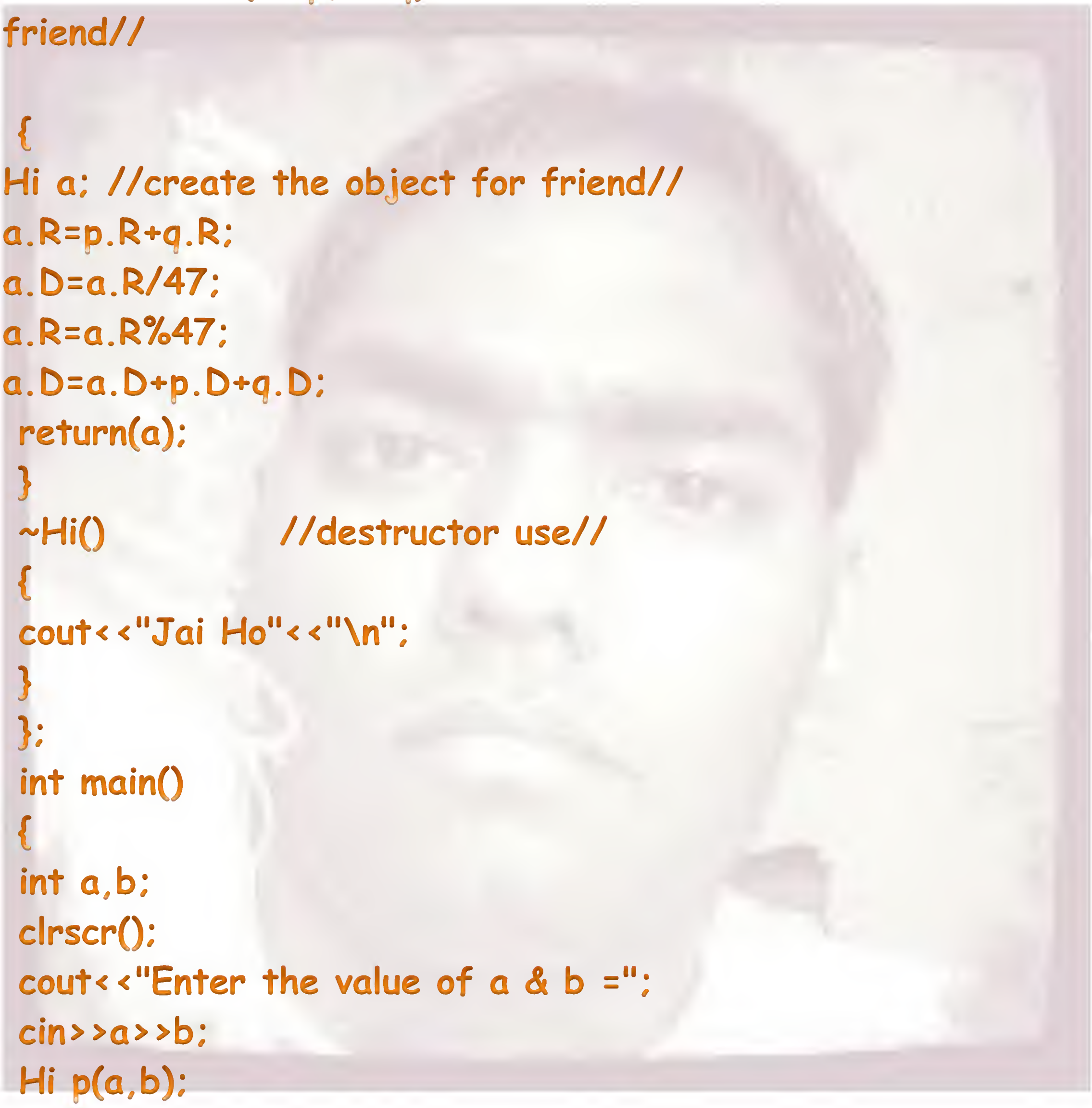
```
#include <iostream.h>
#include <conio.h>
class comp
        {
        int real, img;
        public:
        comp (int a, int b)
        {
        real=a;
        img=b;
        }
        void display()
        {
        cout <<real<<"x,f"<<img<<"i";
        }
        friend comp sum (comp p, comp q) //declare & definition
of friend//
        {
        comp z;   //create the object for friend//
        z.real= p.real+q.real;
        z.img=p.img+q.img;
```

```
      return (z);
      }
      };
      int main ()
      {
      int a, b, c, d;
      clrscr ();
      cout<<"Enter the value of a, b,c, & d=";
      cin>>a>>b>>c>>d;
      comp p (a, b);//create the object and call the
constructor//
      comp q (c, d); //create the object and call the
constructor//
      comp r;
      r= sum (p, q);
      r. display( );
      getch( );
      return(0);
      }
```

Example

```
#include<iostream.h>
 #include<conio.h>
 class Hi
 {
 int D, R;
 public:
 Hi()       //default constructor //
 {

 }
 Hi(int a, int b) //declare & definition of member function//
 {
 D=a;
 R=b;
```



```
 }
 friend void display(Hi M) //declare & definition of  the
friend//
 {
 cout<<"Dollar="<<M.D<<"\n"<<"Rupees="<<M.R<<"\n";
 }
 friend Hi cal(Hi p,Hi q) //declare & definition of  the
friend//

 {
Hi a; //create the object for friend//
a.R=p.R+q.R;
a.D=a.R/47;
a.R=a.R%47;
a.D=a.D+p.D+q.D;
 return(a);
 }
 ~Hi()            //destructor use//
 {
 cout<<"Jai Ho"<<"\n";
 }
 };
 int main()
 {
 int a,b;
 clrscr();
 cout<<"Enter the value of a & b =";
 cin>>a>>b;
 Hi p(a,b);
 Hi q(p);  //copy constructor use//
 Hi i;
 i=cal(p,q);
 display(p);
 display(q);
 display(i);
 getch();
 return(0);            }
```

COPY CONSTRUCTOR

A copy constructor is a constructor of the form class name. The computer will use the copy constructor, whenever you initialize an instance using value of another instance of the since type.

Ex.

```
1) //class name//    comp p (10, 20); // argument//
                     comp q;  // p & q (object)//
                     q=p;
```

First are create the object (q) and then Assign the value of p in q

```
2) //class name//    comp p (10, 20);          // argument//
                     comp q=p;            // p & q (object)//
```

((copy constructor are called here i.e. q=p value of p will be copied in q))

NOTE:

Copy constructor can call in three way

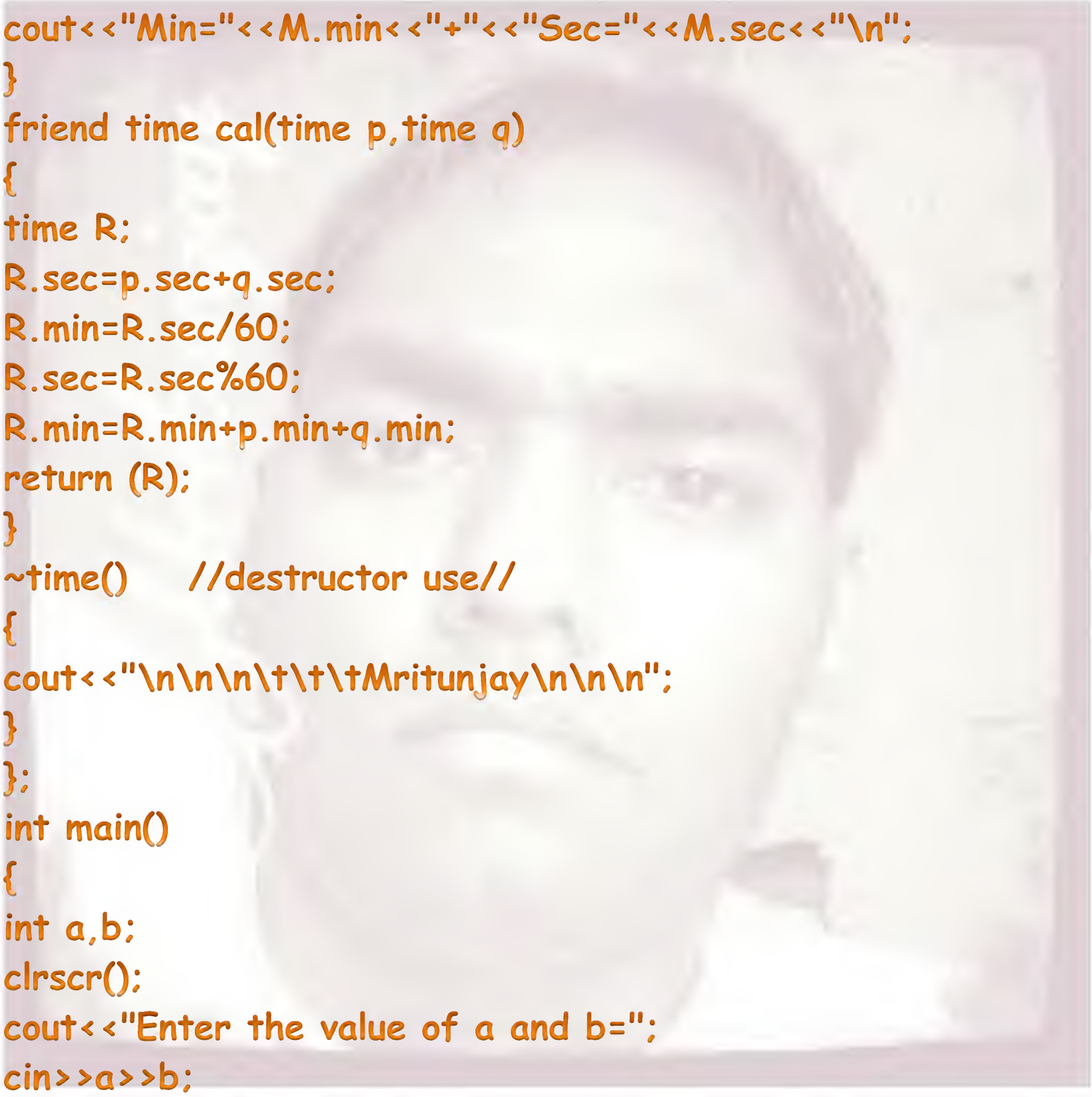
1st way	comp p (10, 20);
	comp q (p);
		2nd way	comp p (10, 20);
			comp q =p;
				3rd way	comp p (10, 20);
					comp q;
					q=p;

Example of Copy Constructor:

```
#include<iostream.h>
#include<conio.h>
class time
{
int min,sec;
public:
time()            //default constructor //

{
}
```

```
time(int a,int b)
{
min=a;
sec=b;
}
friend time showdata(time M)
{
cout<<"Min="<<M.min<<"+"<<"Sec="<<M.sec<<"\n";
}
friend time cal(time p,time q)
{
time R;
R.sec=p.sec+q.sec;
R.min=R.sec/60;
R.sec=R.sec%60;
R.min=R.min+p.min+q.min;
return (R);
}
~time()     //destructor use//
{
cout<<"\n\n\n\t\t\tMritunjay\n\n\n";
}
};
int main()
{
int a,b;
clrscr();
cout<<"Enter the value of a and b=";
cin>>a>>b;
time p(a,b);
time q;
q=p;        //using the copy constructor//3rd way//
time A;
A=cal(p,q);
showdata(p);
```



```
showdata(q);
showdata(A);
getch();
return(0);
}
```

DESTRUCTOR

A destruction function as same name that of class name or construct or name but prefix (~) tilde sign.

A class can have only one destructor. It can not take argument specify a return value OR explicit return value. The overloading of destructor is not possible.

NOTE:

1) It is also special kind of function.
2) It is also same name as that of class name or constructor name but prefix tilde (~) sign

Ex..

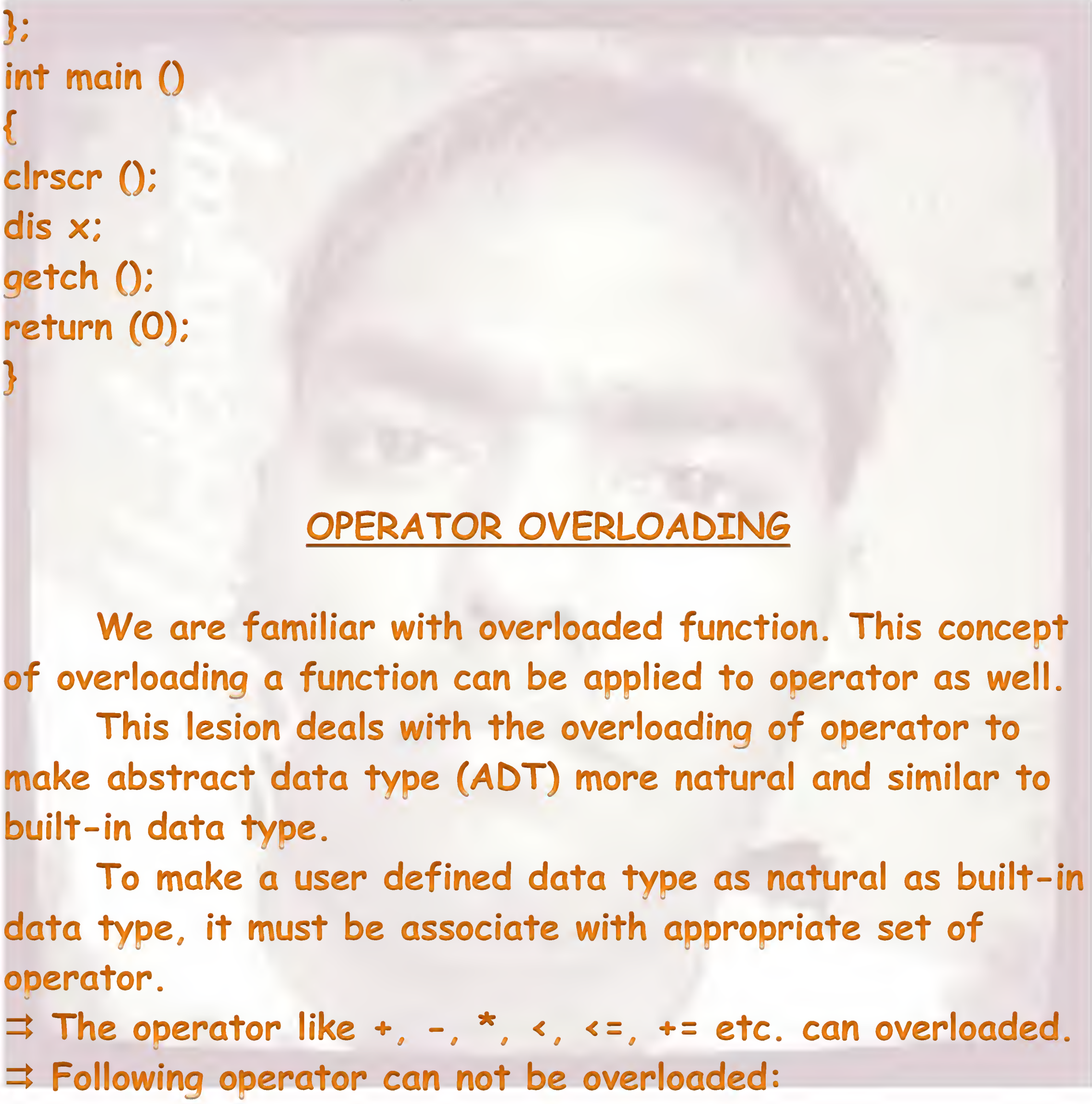
```
~dis()
{
…………
}
```

3) It has no any return value, not even void.
4) It is used for the memory deal location.
5) Not pass the argument.
6) Overloading not passable
7) We not create the destructor then computer create the default destructor.
8) Destructors can also use before closing class.

Example of Destructor:

```
#include<iostream.h>
#include<conio.h>
class dis
{
int v=10;
public:
dis ()      //declare & definition of constructor//
{
cout<< "Number us="<<v;
}
~dis ()          //declare & definition of destructor //
{

}
};
int main ()
{
clrscr ();
dis x;
getch ();
return (0);
}
```



OPERATOR OVERLOADING

We are familiar with overloaded function. This concept of overloading a function can be applied to operator as well.

This lesion deals with the overloading of operator to make abstract data type (ADT) more natural and similar to built-in data type.

To make a user defined data type as natural as built-in data type, it must be associate with appropriate set of operator.

⇉ The operator like +, -, *, <, <=, += etc. can overloaded.

⇉ Following operator can not be overloaded:

a) Class member access operator (. , :).
b) Scope resolution operator (::).
c) Size operator (size of).
d) Conditional operator (?:).

DEFINITION OF OPERATOR OVERLOADING

Operator overloading is defined the additional task of operator.

Operator function must be either:

a) Member function (Non-Static)

b) Friend function

NOTE:

Although the semantics of an operator can executed but we can not changes.

→ Syntax (we cannot change the syntax of operator)

→ Grammatical Rules

→ Number of operand

→ Precedence

→ Associative

RULES FOR OPERATOR OVERLOADING

→ Only existing operators can be overloaded. New operator cannot be created or overloaded.

→ The overloaded operator must have at least one operand that is of user defined type.

→ We can not change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract (-) one value from the other.

→ Overloaded operators follow the syntax rules of original operators. The cannot be overriding.

→ The are same operator that cannot be overloaded (size of, ., *, ::,? :)

→ We cannot used friend function to overload certain operator (=, [], (), - >)

→ Unary operators -), overloaded by means of a member function, take no explicit argument and return no explicit value.

→ Binary operator overloaded through a member function take one explicit argument and those which are overloaded through friend function take two explicit argument.

→ When using binary operator overloaded through a member function, the left hand operand must be an object of the relevant class.

→ Binary arithmetic operator such as "+, -, *, /" must explicitly return a value. They must not attempt to change the their own arguments.

PURPOSE OF OPERATOR OVERLOADING

→ Create a class that defined data type that is to be used in overloading operator

→ Declared the operator function

```
    (+, -, <, >, ++, -- ……...)
    operator op()
```

In the public part of a class as normal member function.

→ Defined the operator function to implement the required operation.

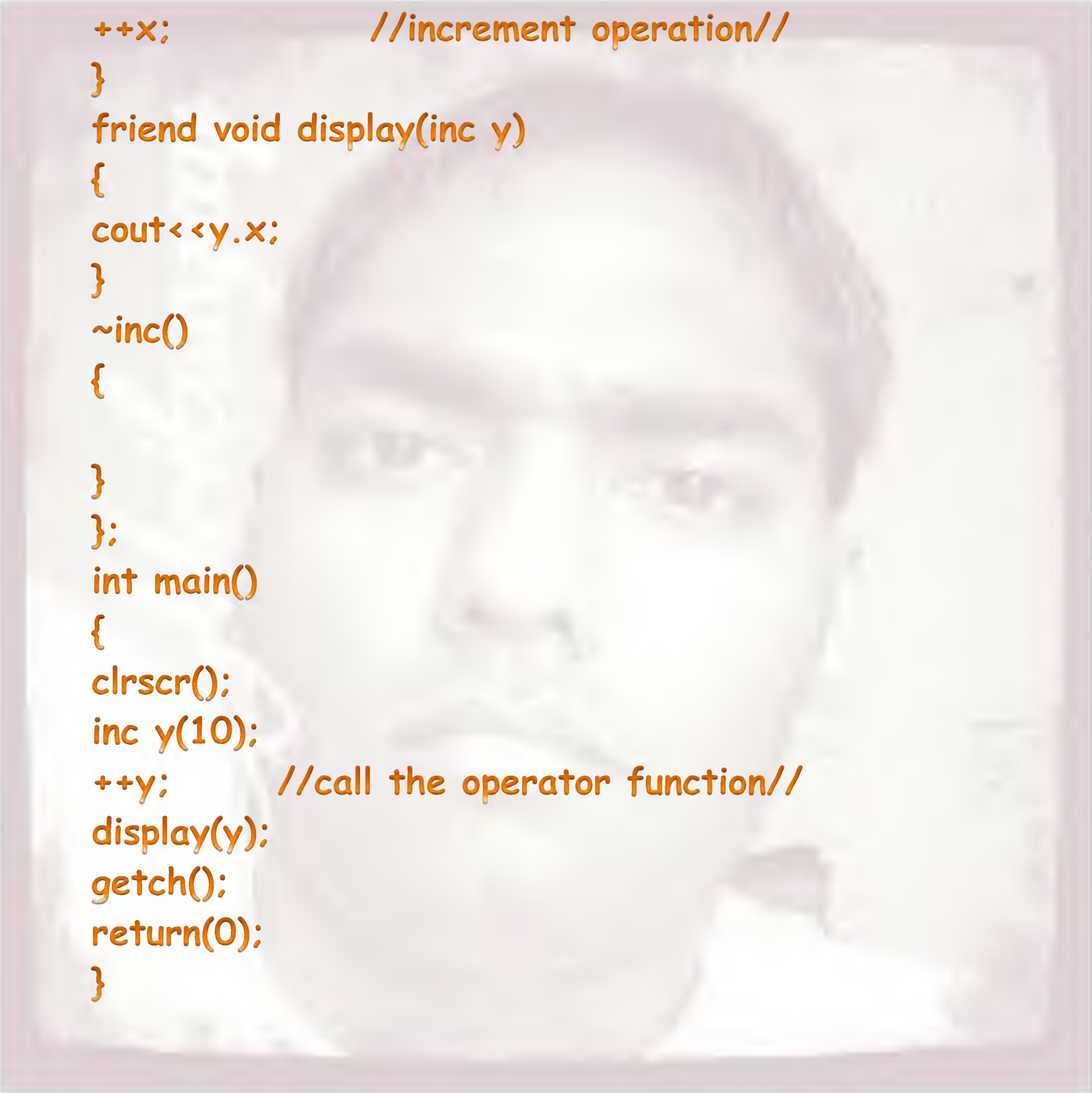
Example of overloading:

1)W A P to overload the ++(increment)operator.

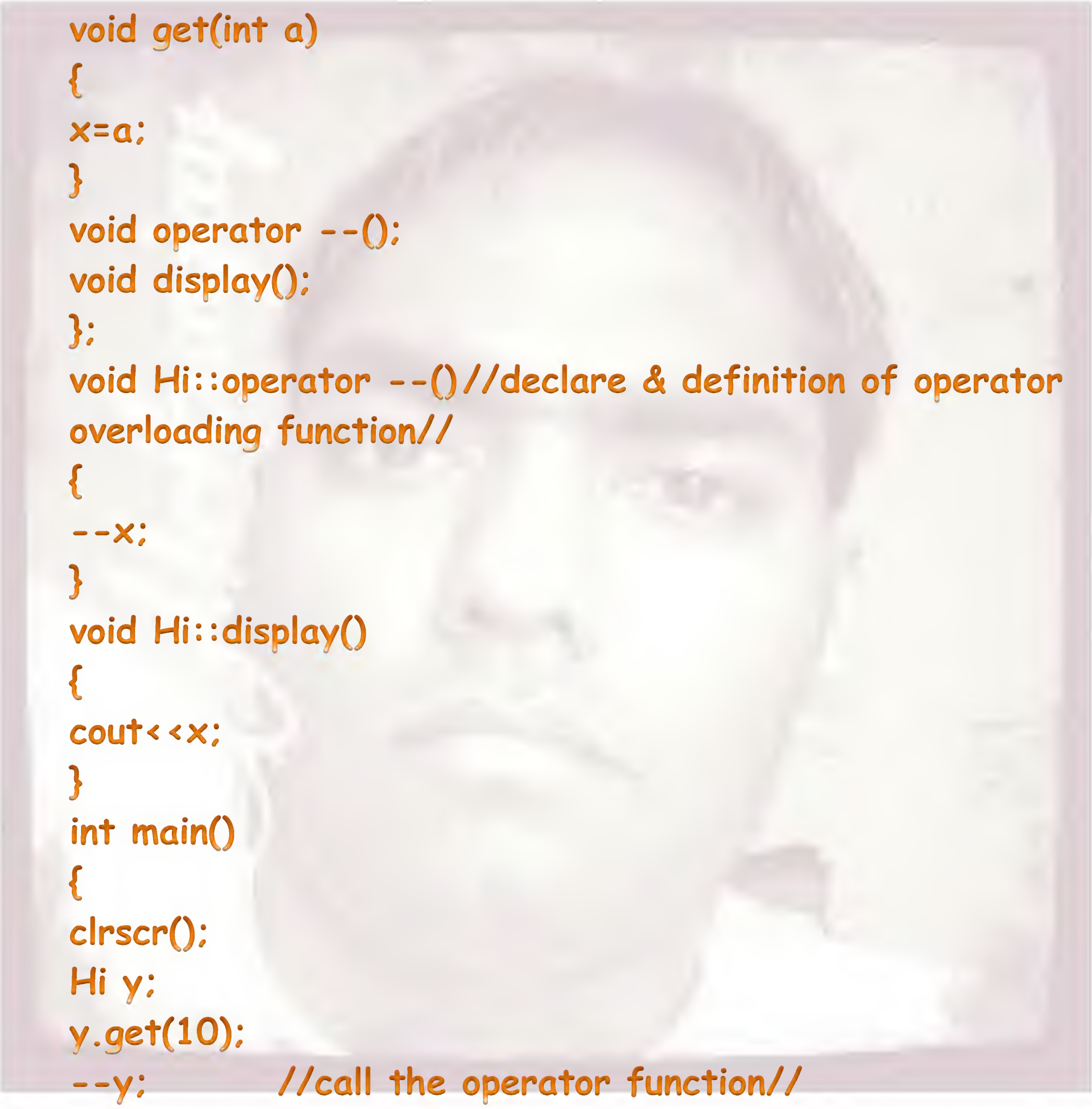
```
    #include<iostream.h>
    #include<conio.h>
    class inc
    {
    int x;
    public:
    inc( )
    {

    }
    inc (int a)
    {
    x=a;
    }
    void operator++( )    //declare & definition of operator
overloading function//
    {
    ++x;               //increment operation//
    }
    friend void display(inc y)
    {
    cout<<y.x;
    }
    ~inc()
    {

    }
    };
    int main()
    {
    clrscr();
    inc y(10);
    ++y;        //call the operator function//
    display(y);
    getch();
    return(0);
    }
```



2)W A P to overload the --(decrement)operator.



```
#include<iostream.h>
#include<conio.h>
class Hi
{
int x;
public:

void get(int a)
{
x=a;
}
void operator --();
void display();
};
void Hi::operator --()//declare & definition of operator
overloading function//
{
--x;
}
void Hi::display()
{
cout<<x;
}
int main()
{
clrscr();
Hi y;
y.get(10);
--y;          //call the operator function//
y.display();
getch();
return(0);
}
```

3) W A P to overload the + (unary plus)operator.

```
#include<iostream.h>
#include<conio.h>
class comp
{
float x;
float y;
public:
comp()
{
}
comp(float real,float img)
{
x=real;
y=img;
}
comp operator +(comp); //declaration  of operator
overloading function//
void display()
{
cout<<x<<"+"<<y<<"i"<<"\n";
}
~comp()
{
cout<<"good"<<"\n";
}
};
comp comp::operator+(comp c)//definition of operator
overloading function//
{
comp temp;
temp.x=x+c.x;
temp.y=y+c.y;
return(temp);
}
int main()
```

```
{
float a,b,c,d;
clrscr();
cout<<"Enter the two imaginary number =";
cin>>a>>b>>c>>d;
comp c1;
comp c2;
comp c3;
c1=comp(a,b);
c2=comp(c,d);
c3=c1+c2;        //call the operator function//
c1.display();
c2.display();
c3.display();
getch();
return(0);
}
```

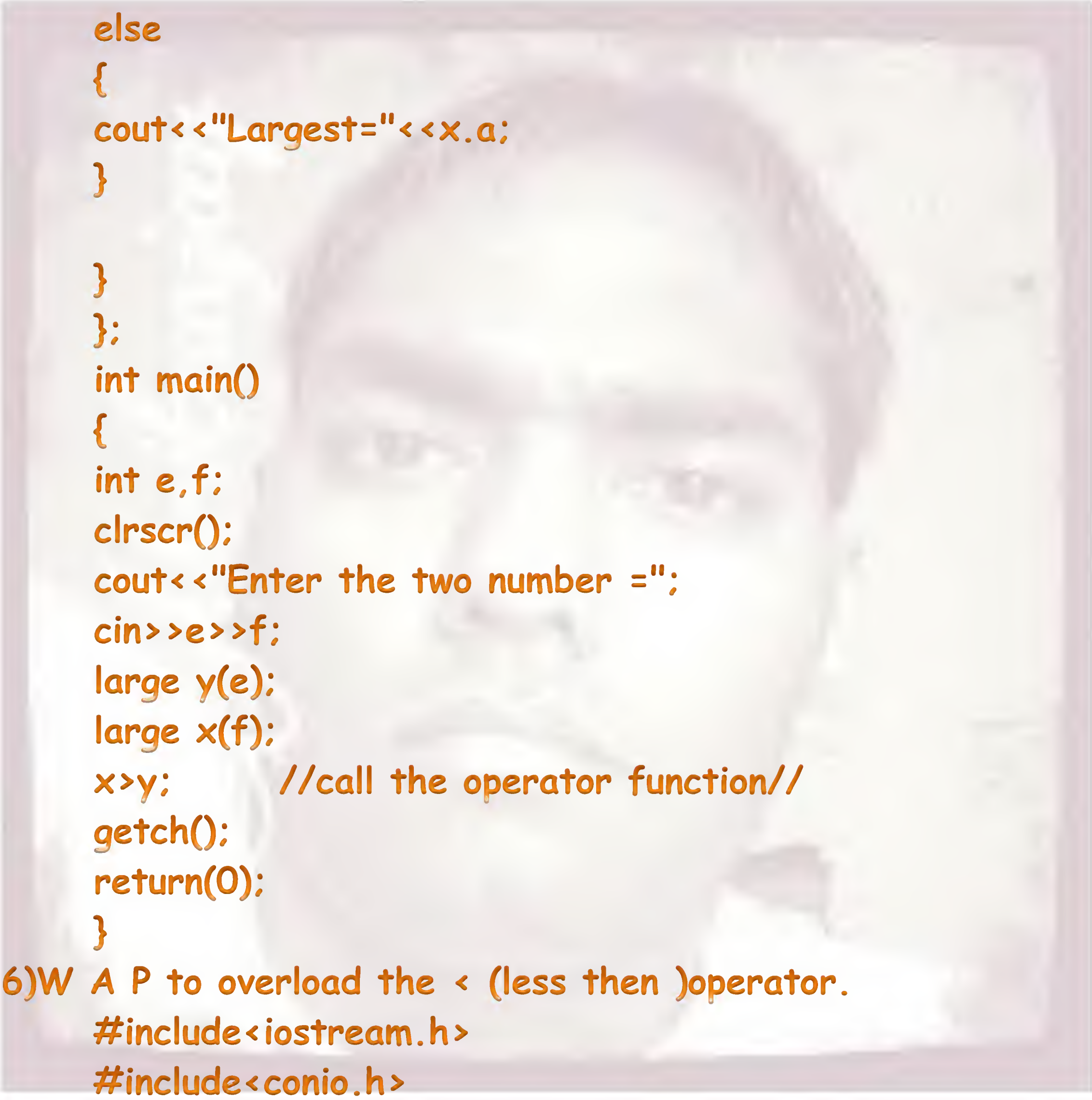
4)W A P to overload the - (unary mines)operator.

```
#include<iostream.h>
#include<conio.h>
class RBK
{
int x;
int y;
int z;
public:
void getdata(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void display()
{
```

```
cout<<x<<"\n"<<y<<"\n"<<z<<"\n";
}
void operator -();
};
void RBK::operator -()
{
x=-x;
y=-y;
z=-z;
}
int main()
{
clrscr();
RBK s;
s.getdata(10,-45,67);
s.display();
-s;          //call the operator function//
s.display();
getch();
return(0);
}
```

5)W A P to overload the > (greater then)operator.

```
#include<iostream.h>
#include<conio.h>
class large
{
int a;
public:
large()
{
}
large(int c)
{
a=c;
```

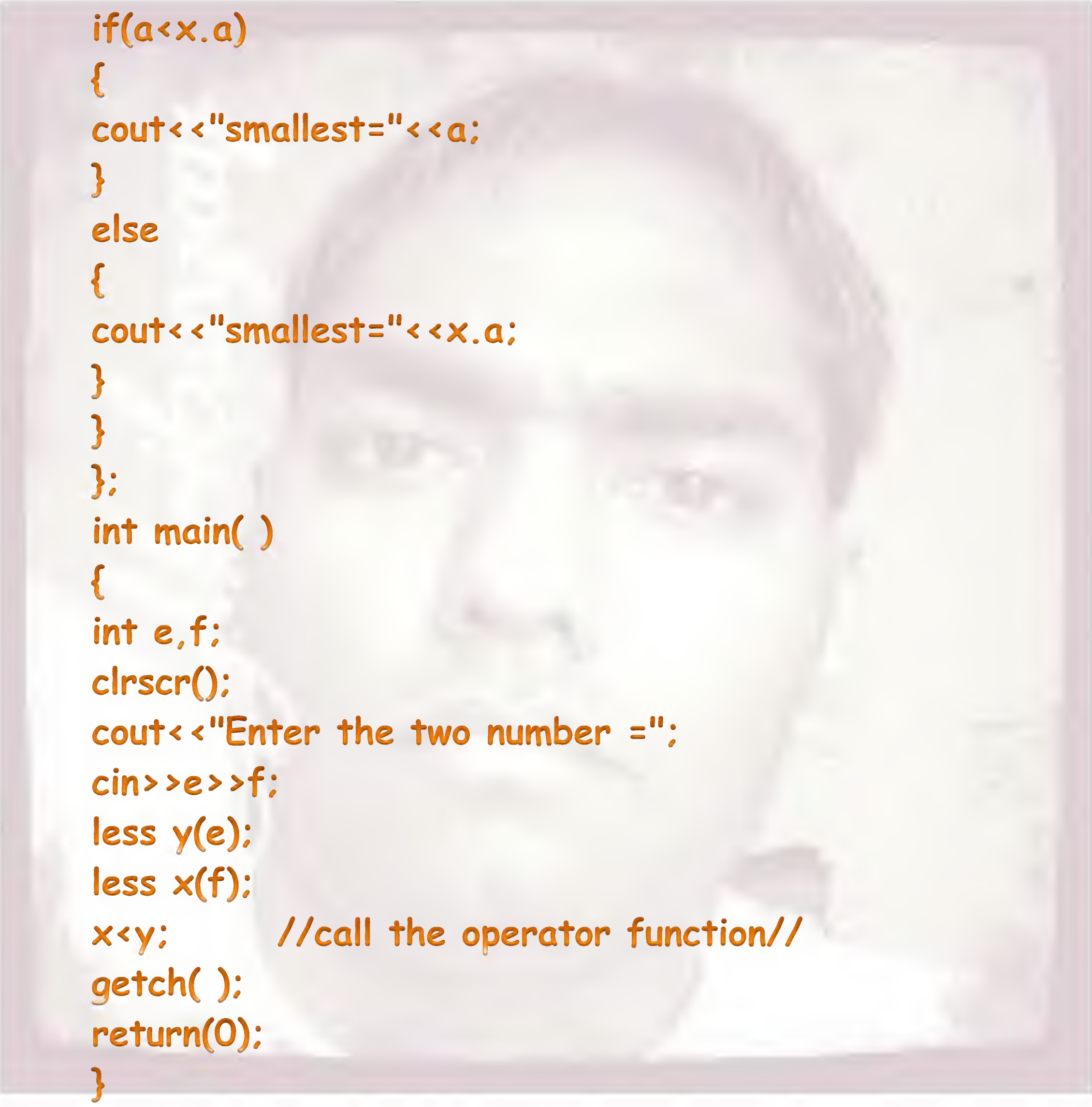


```
}
large operator >(large x)
{
if(a>x.a)
{
cout<<"largest="<<a;
}
else
{
cout<<"Largest="<<x.a;
}

}
};
int main()
{
int e,f;
clrscr();
cout<<"Enter the two number =";
cin>>e>>f;
large y(e);
large x(f);
x>y;        //call the operator function//
getch();
return(0);
}
```

6)W A P to overload the < (less then)operator.

```
#include<iostream.h>
#include<conio.h>
class less
{
int a;
public:
less( )
{
```

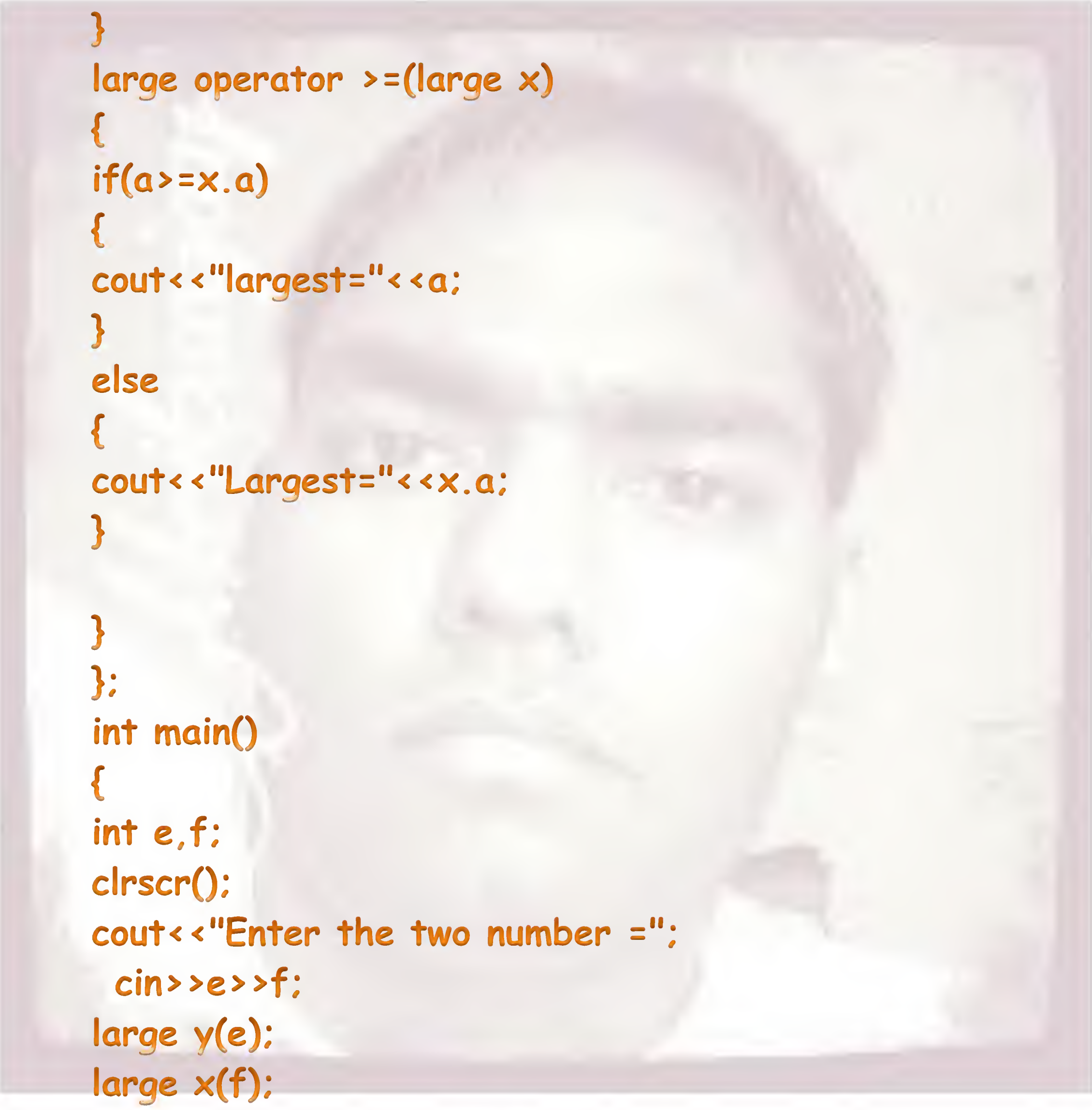


```
}
less(int c)
{
a=c;
}
less operator <(less x)
{
if(a<x.a)
{
cout<<"smallest="<<a;
}
else
{
cout<<"smallest="<<x.a;
}
}
};
int main( )
{
int e,f;
clrscr();
cout<<"Enter the two number =";
cin>>e>>f;
less y(e);
less x(f);
x<y;        //call the operator function//
getch( );
return(0);
}
```

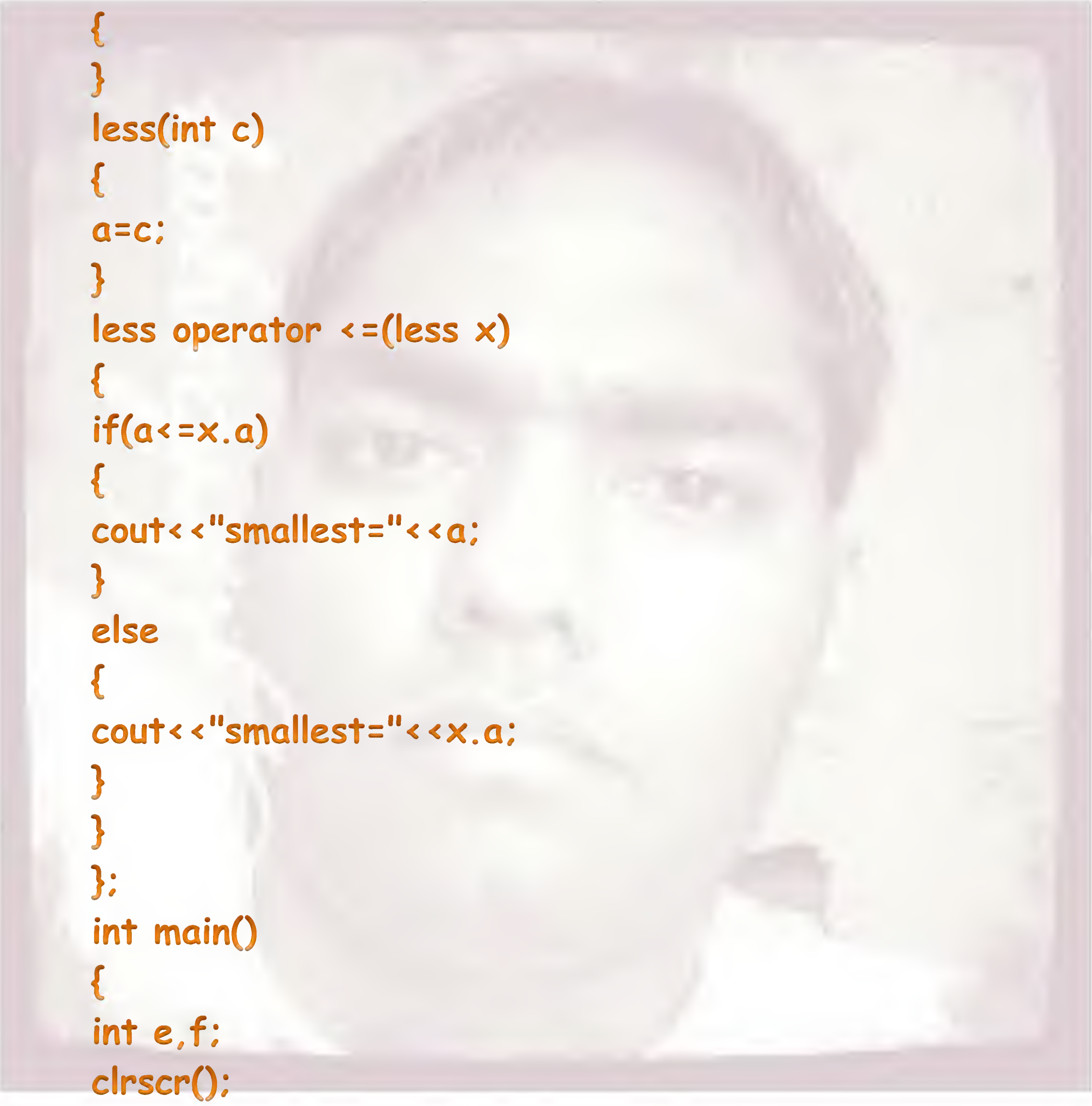
7)W A P to overload the >= (greater then equal)operator.

```
#include<iostream.h>
#include<conio.h>
class large
{
int a;
public:
large()
{
}
large(int c)
{
a=c;
}
large operator >=(large x)
{
if(a>=x.a)
{
cout<<"largest="<<a;
}
else
{
cout<<"Largest="<<x.a;
}

}
};
int main()
{
int e,f;
clrscr();
cout<<"Enter the two number =";
  cin>>e>>f;
large y(e);
large x(f);
x>=y;             //call the operator function//
getch();
return(0);
}
```



8)W A P to overload the <= (less then equal)operator.



```
#include<iostream.h>
#include<conio.h>
class less
{
int a;
public:
less()
{
}
less(int c)
{
a=c;
}
less operator <=(less x)
{
if(a<=x.a)
{
cout<<"smallest="<<a;
}
else
{
cout<<"smallest="<<x.a;
}
}
};
int main()
{
int e,f;
clrscr();
cout<<"Enter the two number =";
cin>>e>>f;
less y(e);
less x(f);
x<=y;            //call the operator function//
getch();
return(0);
}
```

INHERITANCE

Inheritance is one of the most important notions of OOP. Inheritance allows creating class from existing class through an IS-A relationship. When members (data and/or function) are inherited, they can be used in the derived class as if they are member of the derived class that inherits them, provided, the base class permits the derived class to do so. If class B inherits from class A, then B is called the specialization of generalized class A. Class A is called the base or super class and class B is called the derived or subclass. A derived class inherits the properties that include data and function members of its base class. Thus, all the code, which is written to work with object of the base class, will work with the objects of derived classes.

class a → Base class
→ Inheritance
class B → Derived class

⇒ Inheritance has important advantage that is most importantly it permits code reusability.

Once a base class written and debug (compile) it need not be touch again but can never the less be adopted to word in different solution.

⇒ Reusing Existing code save time and money and increasing the program reliability.

➧Access identifier of a class.

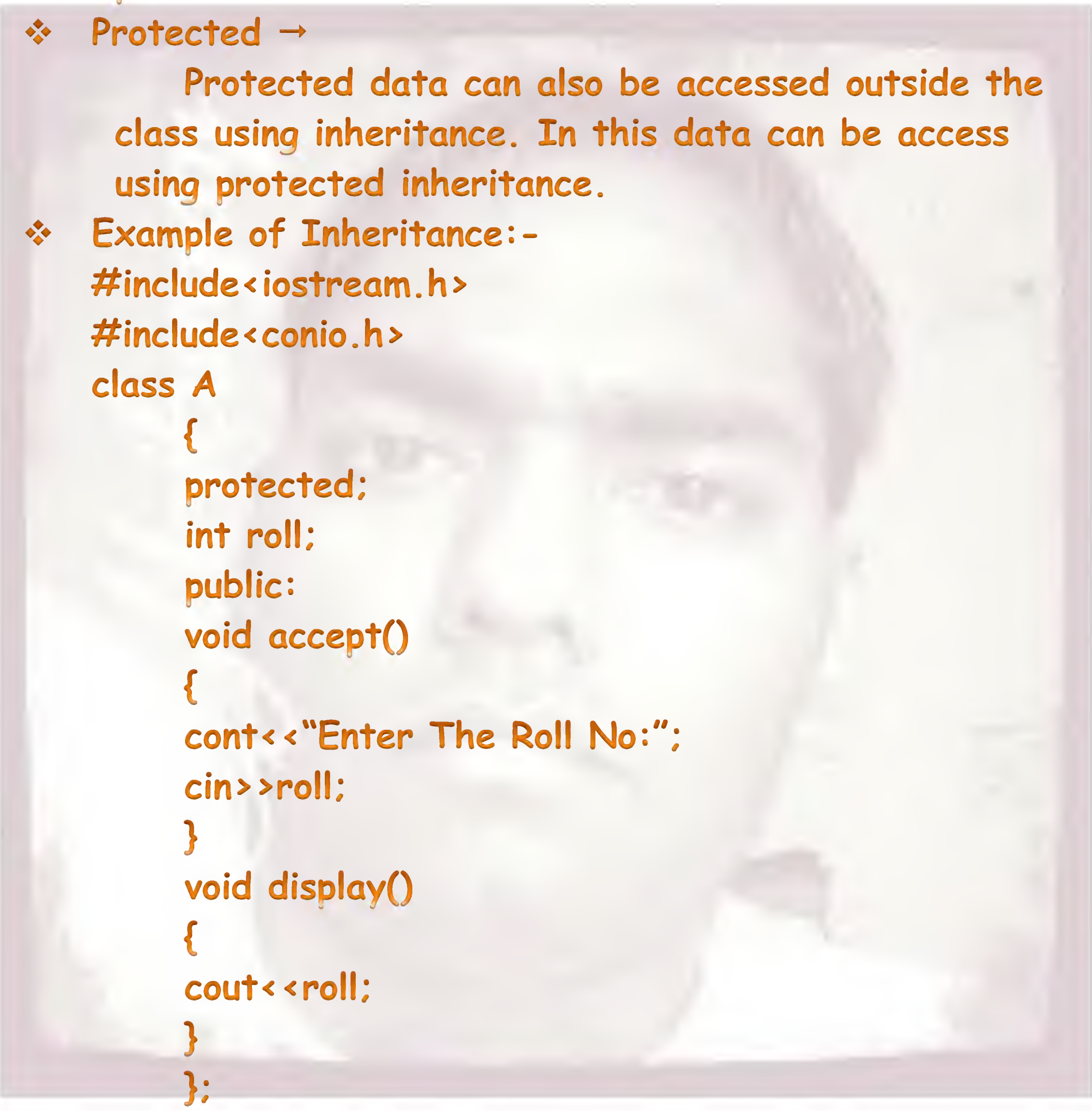
1) Private → Private never inheritance.
2) Public → con be inheritance
3) Protected → can be inheritance

❖ Private →

Private data can not be accessed outside the class except number function private data can not be inherited.

- Pubic →

  Public data or member function can easily to be access outside the class. It can be inheritance using public inheritance.



- Protected →

  Protected data can also be accessed outside the class using inheritance. In this data can be access using protected inheritance.

- Example of Inheritance:-

```
#include<iostream.h>
#include<conio.h>
class A
        {
        protected;
        int roll;
        public:
        void accept()
        {
        cont<<"Enter The Roll No:";
        cin>>roll;
        }
        void display()
        {
        cout<<roll;
        }
        };
        class B:  public A
        {
        char name [21];
        public:
        void accept2()
        {
```

```
accept2 ();
cout<<"Enter The Name";
cin>>name;
}
void display2 ()
{
display2 ();
cont<<name;
}
};
int main()
{
A x;
B y;
x.accept ();
x.display();
y.accept2();
y.display2();
getch();
return(0);
}
```

Kind of Inheritance

1st - Single Inheritance [class B: public A]

A Base class

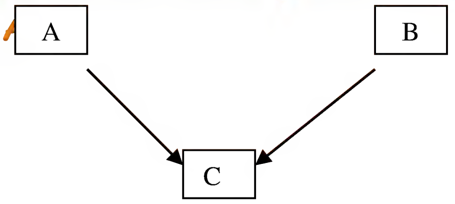
↓

B Derived class

A derived class with only one base class is called Single Inheritance.

2nd - Multiple Inheritance [class c: public A, public B]

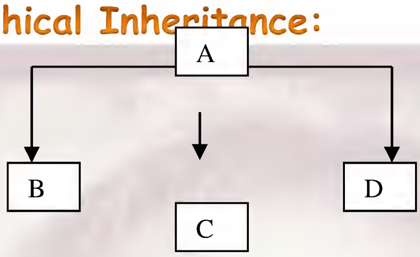
Base class A Base Class B



Derived class C
Which Inheritance of class A
and B

⇒ A derived class, one with several base classes is called multiple inheritances.

3rd – Hierarchical Inheritance:



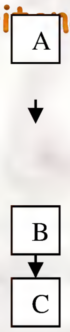
Base class A

Derived class B
Derived class
D

Derived class C

⇉ The traits of one class may be inherited by more then one class is called hierarchical inheritance.

4th – Multiple Inheritance:



(class: public A)
(class c: public B)

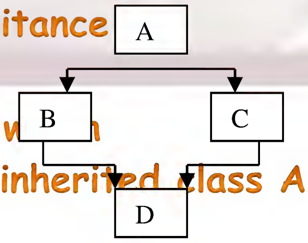
Base class A

Derived class B and
inherited class

Derived class C and
inherited class A, class B

⇒ The mechanism of periving a class from another derived class B known as M.L.g.

5th – Hybrid Inheritance



Base class A

Derived class B with
which inherited inherited class A
class A

Derived class C

Derived class d, which
inherited class A,

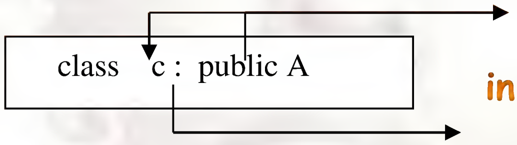
Class B, and class C.

⇒ Collection of multilevel and multiple inheritance is called Hybrid Inheritance.

:FACTS:

1) A derived class inherit all the capabilities of A base class, but can add new factures of its own. By mapping these addition the base class remain unchanged.
2) Protected member behave just like private member until a new class is derived from the base class and that has protected member.
3) If a base class has a private member, those members are not accessible to the derived class. Protected member are public to derived class but private to rest of the program.
4) A derived class classify that a base class is public or private by using the notation.

➪In the declaration of the derived class



Public portion A is inheritance

Symbol for inheritance or (:) column it the inheritance

Class is a derived class where as class A is the base class and class C inherited only the public data of base class C inherited only the public data of base class A into derived class C:

class c: private A ×

class c: protected ✓

⇒ When we define an object of a derived class the compiler execute the contractor function of the box class followed by constructor function of the derived.

The parameters list of or the derived class contractor function may be different from that of the base class constructor function.

⇒ When a base class and derived class have public member function with the same name and parameter the function in a derived class get a priority when the function is called as a member of the derived class object.

⇒ A program can declare object of both the base and derived class object. These two object are indenedent with each other.

Type of Inheritance

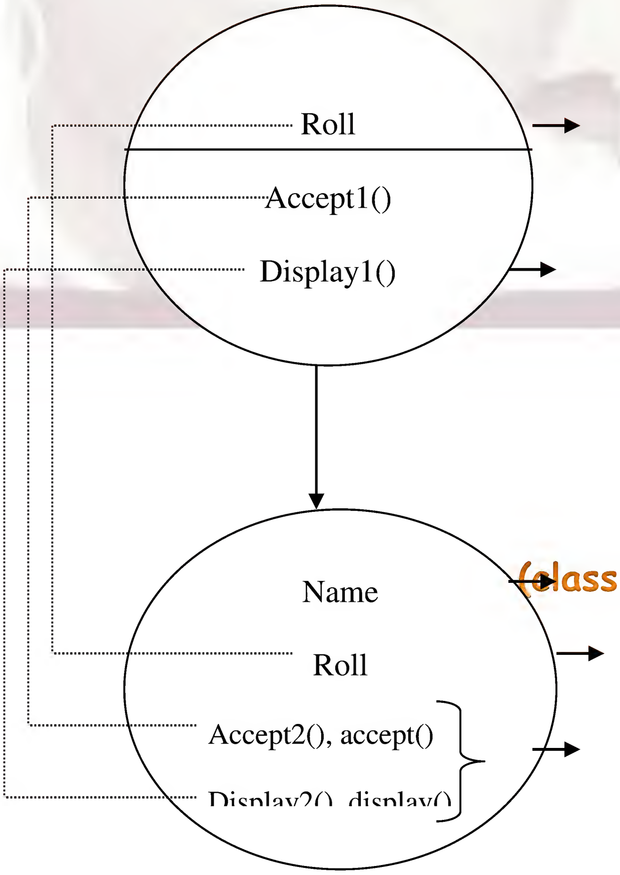
1st - Public Inheritance

2nd- Protected Inheritance

1st - Public Inheritance

1) Private member are not inherited as usual.
2) Protected member become protected in the derived class.
3) Public member become public in the derived class.

Class A (Base close)



Private class of A

Public of class A

(class B: public A)

Class B (derived class)

Private of class B

Protected

Public portion of class B

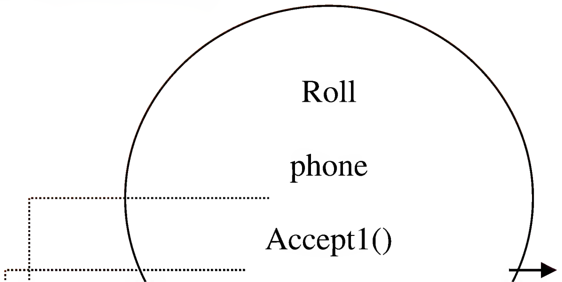
Example of Public Inheritance :

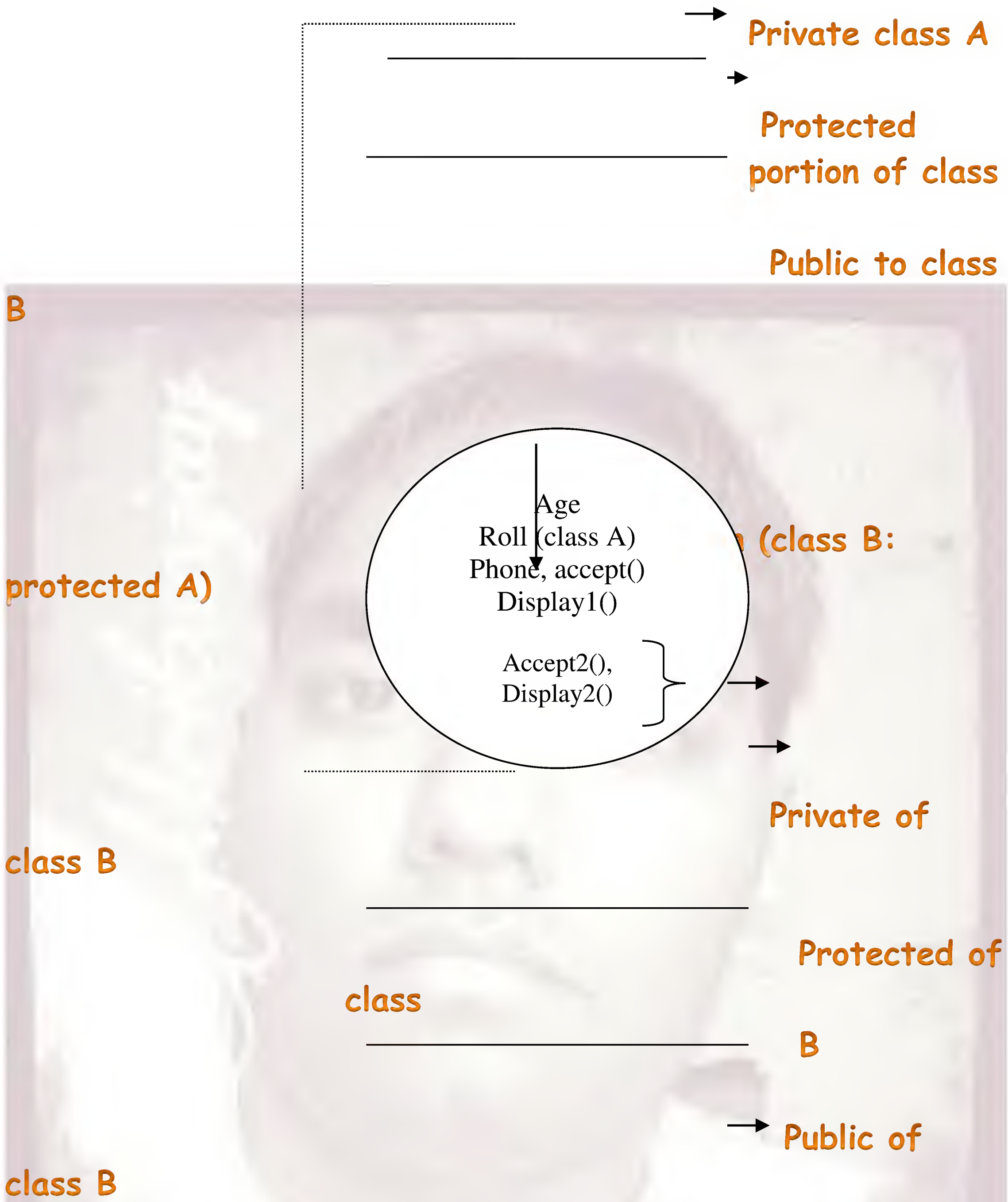
W.A.P. to accept the first no. from class A and second no. accept by class B & display first no. and second no. through class B.

```
#include<iostream.h>
#include<conio.h>
class a
{
int n;
public:
void accept()
{
cout<<"\n\n\nEnter the first number:\t ";
cin>>n;
}
void showdata()
{
cout<<"\n\n\n    number of a class=\t"<<n;
}
};
class b:public a      // Public Inheritance of class A//
{
int m;
public:
void getdata()
{
cout<<"\n\n Enter the second number:\t ";
cin>>m;
accept();
}
void display()
{
showdata();
cout<<"\n\n second number is=\t"<<m;
}
};
int main()
{
clrscr();
a x;
b y;
x.accept();
x.showdata();
y.getdata();
y.display();
getch();
return(0);
}
```

2nd - Protected Inheritance

1) Private member are not inherited as usual or always.
2) Protected member will become protected in new class or derived class.
3) Public member will also become protected in the new class or derived class.




Private class A
Protected portion of class
Public to class B
Age
Roll (class A)
Phone, accept()
Display1()
Accept2(),
Display2()
(class B:
protected A)
Private of class B
Protected of class B
Public of class B

HYBRID INHERITANCE

Example:-

```
#include<iostream.h>
#include<conio.h>
class A
{
int p;
public:
};
class B:public A
{
int p;
public:
};
class C:public A,public B
{
public:
void accept()
{
cout<<"Enter the value of p for class A=";
cin>>A::p;                              //This is used for remain the
ambiguity //
cout<<"Enter the value of p for class B=";
cin>>B::p;                         //This is used for remain the
ambiguity //
}
void display()
{
cout<<A::p;                                   //This is used for
remain the ambiguity //
cout<<B::p;                                   //This is used for
remain the ambiguity //
}
};
int main()
{
clrscr();
C x;
x.accept();
x.display();
getch();
return(0);
}
```

NOTE:-

- ✓ In ambiguity condition scope resolution operator is used for particular data.
- ✓ Suppose we have an ambiguity condition of a class to another class (multiple data are in ambiguity condition) than we have needed to use virtual class.

VIRTUAL CLASS

⇒ Virtual class is also known as virtual base class. It is always to make virtual of the base class in ambiguity condition.

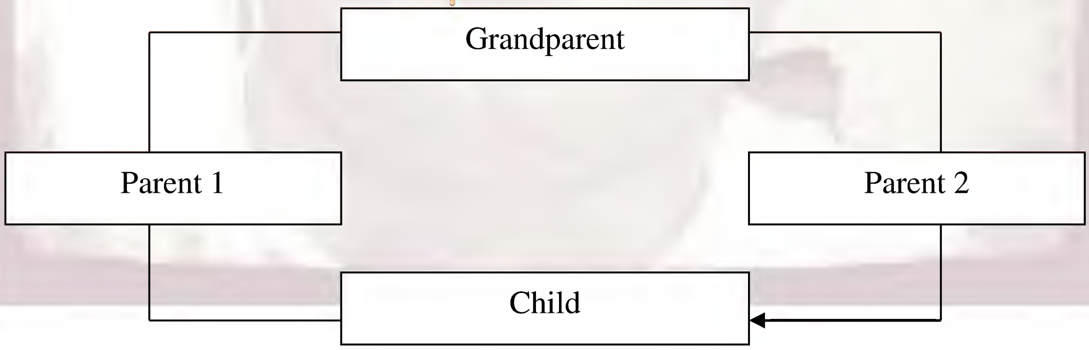
NOTE:-

- ✓ Virtual class is always come with the base class only with keyword virtual.
- ✓ Virtual base class can be writing either for public, protected, private in inheritance.
- ✓ Virtual class or virtual function is always with is a base class.

Condition for Virtual Class

⇒ Virtual class is also know as virtual base class. It is always to make virtual of the base class in ambiguity condition.

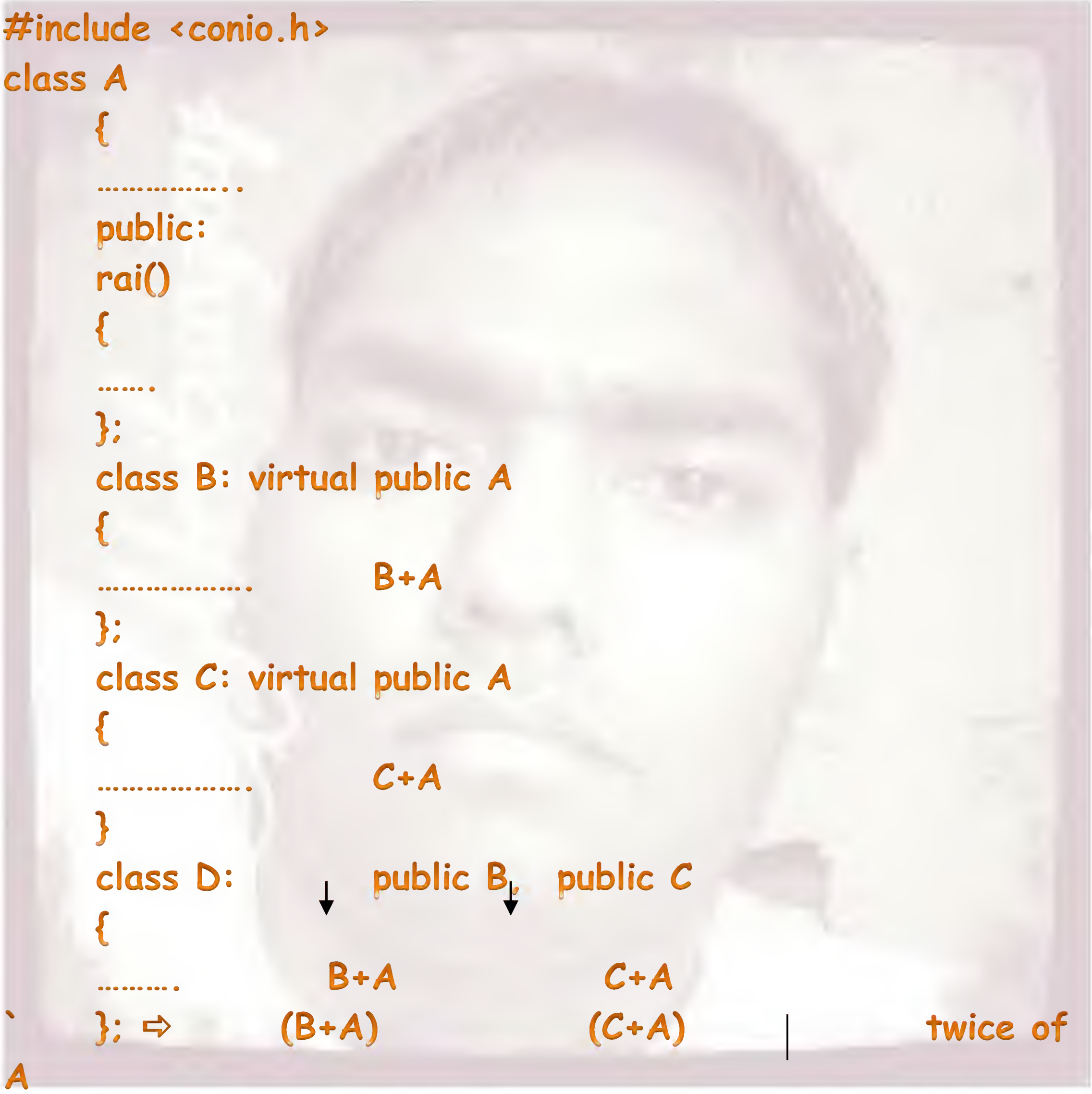
Multi path Inheritance



⇒ All the public and protected member of grandparent are inherited into child twice, first via parent and second via parent2 this introduce ambiguity and should be avoided

⇒ These multi path can be avoided by making the common base class as virtual base class.

Standard format for Virtual Class

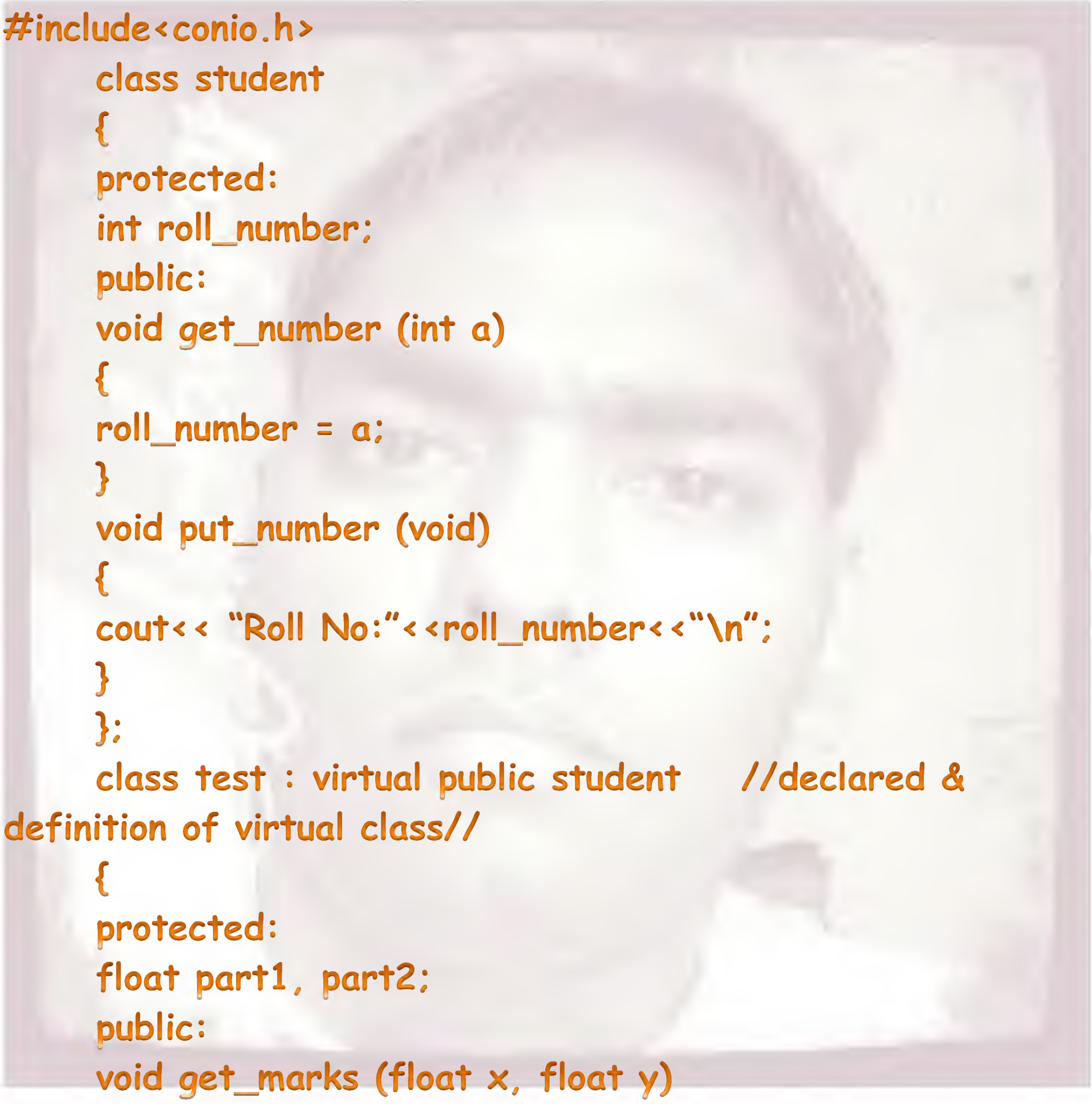


```
#include <iostream.h>
#include <conio.h>
class A
      {
      ……………..
      public:
      rai()
      {
      …….
      };
      class B: virtual public A
      {
      ……………….        B+A
      };
      class C: virtual public A
      {
      ……………….        C+A
      }
      class D:        ↓    public B,  ↓ public C
      {
      ………         B+A               C+A
`     }; ⇨       (B+A)             (C+A)     |         twice of
A
                    ↑             ↑

int main ()
      {
      A x;
      B y;
      C z;
```

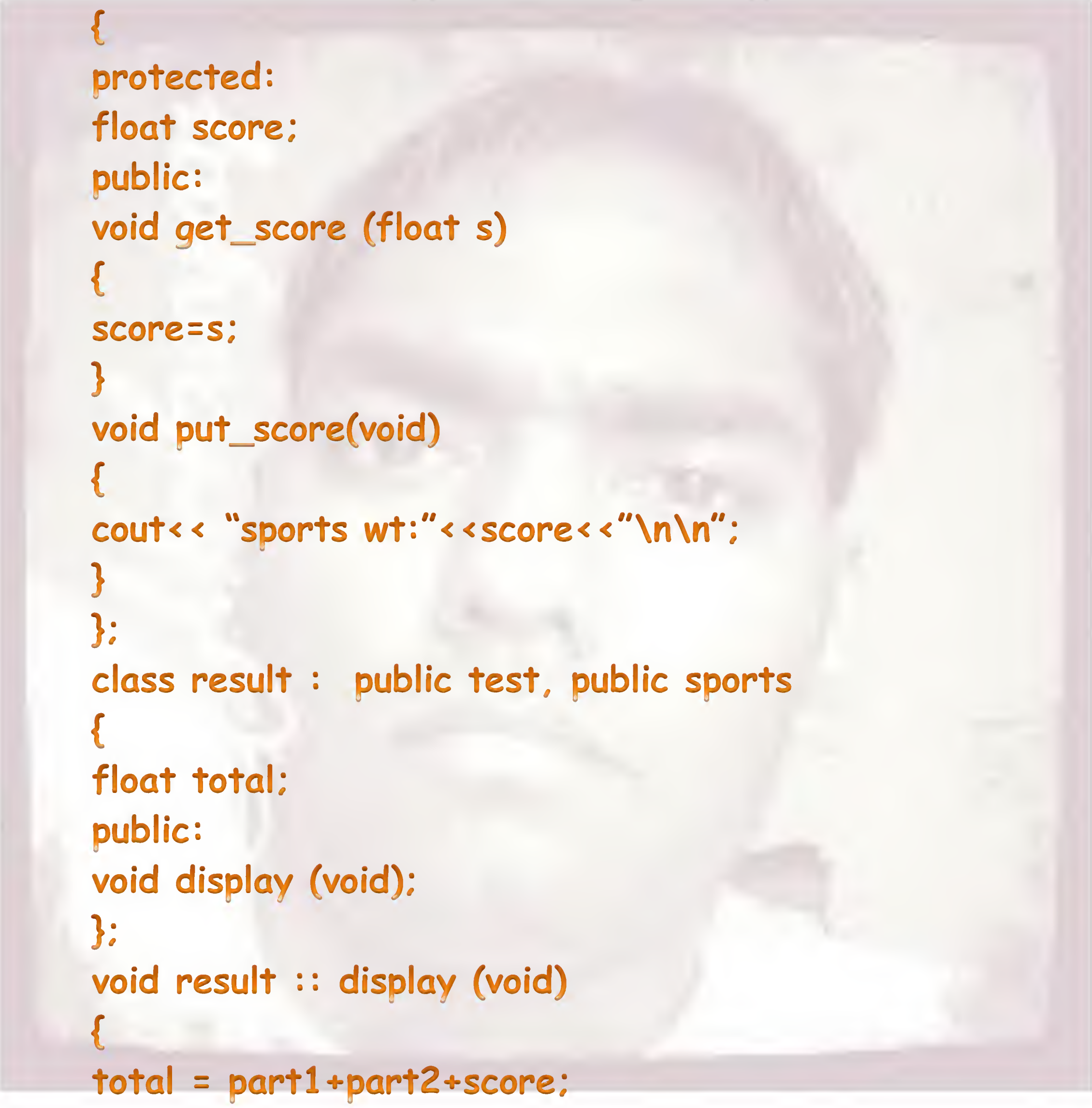
```
        D y;
        return (0);
        }
```

Example of Virtual Class:



```
#include<iostream.h>
#include<conio.h>
        class student
        {
        protected:
        int roll_number;
        public:
        void get_number (int a)
        {
        roll_number = a;
        }
        void put_number (void)
        {
        cout<< "Roll No:"<<roll_number<<"\n";
        }
        };
        class test : virtual public student     //declared &
definition of virtual class//
        {
        protected:
        float part1, part2;
        public:
        void get_marks (float x, float y)
        {
        part1=x; part2=y;
        }
        void put_marks (void)
        {
```

```
      cout << “Marks obtained:”<<“\n” << “part1=”<<
part1<< “\n”
              << “part2=” << part2<< “\n”;
      }
      };
      class sports : public virtual student  //declared &
definition of virtual class//
      {
      protected:
      float score;
      public:
      void get_score (float s)
      {
      score=s;
      }
      void put_score(void)
      {
      cout<< “sports wt:”<<score<<”\n\n”;
      }
      };
      class result :  public test, public sports
      {
      float total;
      public:
      void display (void);
      };
      void result :: display (void)
      {
      total = part1+part2+score;
      put_number();
      put_marks();
      put_score();
      cout<<”Total Score:”<<total<< “\n”;
      }
      int main ()
```



```
{
clrscr();
result student_1;
student_1.get_number(678);
student_1.get_marks (30.5, 25.5);
student_1.get_score (7.0);
student_1.display();
getch();
return (0);
}
```

Virtual Function:-

Ex.

```
            class  A
            {
            ……………..
            public:
            void display()
            {
            ……………..
            }
            ………………..
            };
//it has also a no.   class B: public A     //Having member
function display()// function display ().   {
            ……….
            public C:
            void display()
}
………………..
}
………………….
};
```

⇒ When we use the same function name in both base class and derived class the function in base class is declared as virtual. Using keyword virtual.

⇒ Preceding (before, prefix) as normal declaration or normal function declaration.

⇒ C++ determine which function to b used at run time based an type of object pointed to by the base pointer.

NOTE:-

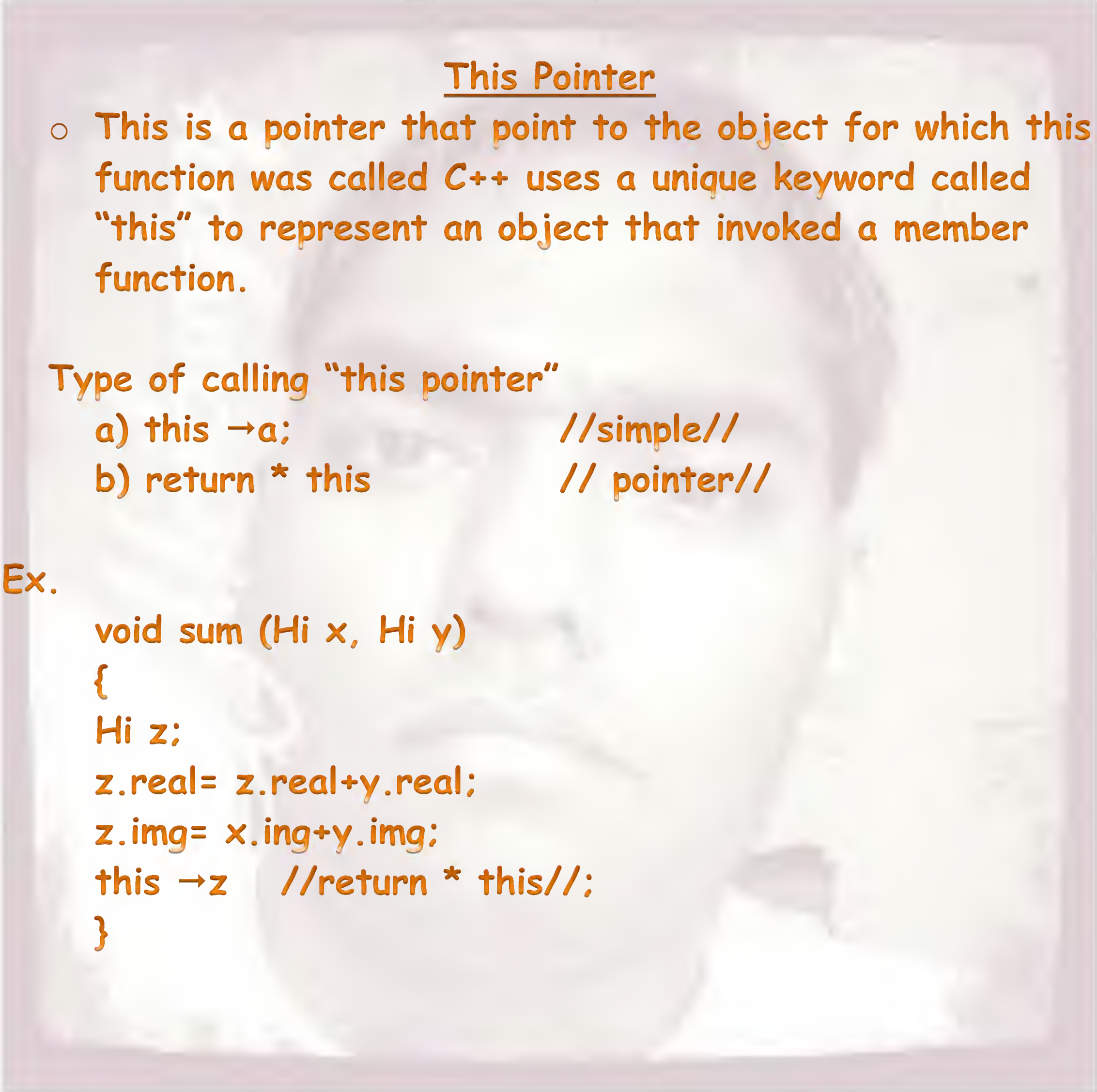
❖ By making the base pointer to point to different version of virtual function

Rules for Virtual Function

1) Virtual function must be a member of same class (member more than one class).
2) They can not be static.
3) They are accessed by using object pointer (b ptr).
4) Virtual function can be friend function of another class.
5) A virtual function in base class must be defined Even though may not be used.
6) We can not have virtual constructor but be can have virtual destructor.
7) While a base pointer can point to any type of derived object the reveres is not true. That is to say we can not use a pointer to a derived class to access on
8) Object of a base class
9) If a Virtual function is defined in the base it need not necessarily redefined in the derived class. In such case coll invoked the base class function.

Pure Virtual Funtion

A Virtual function equated to zero is call pure virtual function. It is in a function declaration in base class that has no definition in base class that has no definition relative to the base class. A class containing such pure function is called abstracted class.



This Pointer

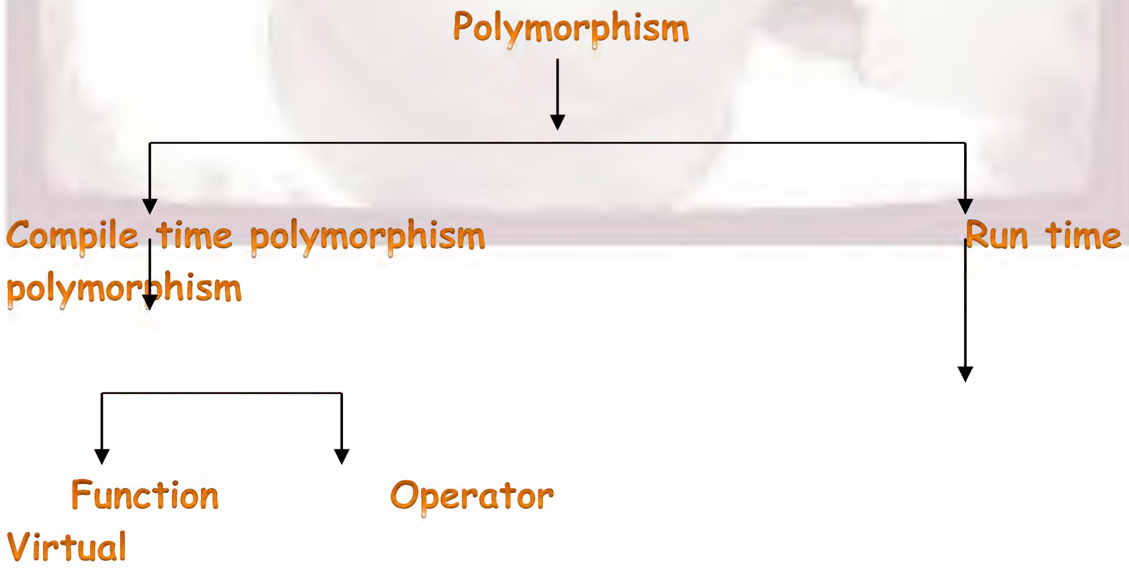
- This is a pointer that point to the object for which this function was called C++ uses a unique keyword called “this” to represent an object that invoked a member function.

Type of calling “this pointer”

```
a) this →a;                  //simple//
b) return * this             // pointer//
```

Ex.

```
void sum (Hi x, Hi y)
{
Hi z;
z.real= z.real+y.real;
z.img= x.ing+y.img;
this →z    //return * this//;
}
```



Polymorphism

Compile time polymorphism

Run time

polymorphism

Function

Operator

Virtual

overloading   overloading
function

Polymorphism

Polymorphism is another important OOPS concept it is a Greek word which means ability to take more than one form or multiple form from sing data.

OR

Polymorphism simple means one name having multiple form.

Compile time polymorphism

Function and operator overloading are the Ex of compile time polymorphism. The overloaded member function selected for invoking (called) by matching arguments of both type and number. The compiler known this information at compile time their fore compiler is able to select the appropriate function for particular call at the compile time itself. This is also called early banding or static banding. It means an object in bounded to a its function call at compile time.

Static Banding

When a function call is resolved at compile time this process is called static banding.

Ex. Function overloading.

Class

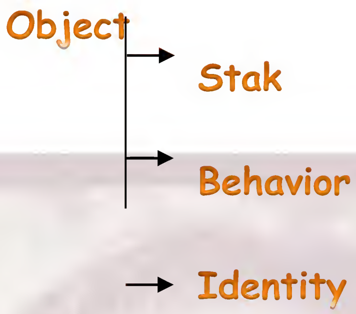
A class is set of object that share a common structure, common behaves and common semantic. It single object is simply an instance of a class.

Object

- ❖ An object is a tangible intenty that exhibits same well defined beavers.
- ❖ An object is any of the following-
    1) A tangible and/or variable.

2) Some thing that may be comprehended interactually

3) Some thing to word which thought or action & directed object.



Object

Stak

Behavior

Identity

Templates

⇒ It is used to defined the data type at the run time. Templates can be used to create a family of classes or function (overloading).

For Ex. A class template for an array class would enable us such as int array, float array, char array and so no.

⇒ Template classes and function eliminate the code duplication for different type (diff. data type) and thus make the program development and easily and more manageable.

Characteristic of Templates

⇒ C++ support the mechanism known as template to implement the concept of Generic Programming.

⇒ Template allow us to general family of class or family of function to handle different data type.

⇒ Template classes and functions eliminate code duplication for different type.

⇒ We can use multiples parameter in both class template and function template.

⇒ In specified class created from a class template is called template class and the process of creating a template class is known as instantiation.

⇒ Like other function template function overloaded.

⇒ Member function of a class template must be defind as function template using the parameter the class template.

⇒ We may also use non type parameter or default function. Such basic or derive data types as arguments template.

NOTE:-

Template can be creating before in main function.

Template < class T>

Template data type (T)

Keyword of template creating a template for class

EXCEPTION HANDLING

Exception handling was not part of the original c++. It has a new feature added in ANSI C++. But today, almost compliers support this feature .

Exception handling provide a type –self integrated approach, for coping with the unusual predictable problems that arise while executing a program.

Type of exception handling

These are two type:-

1st: Synchronous exception.

2nd: Asynchronous exception.

ASYNCHRONOUS EXCEPTION

The error that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions.

SYNCHRONOUS EXCEPTION

The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

The purpose of the exception handling mechanism is to provide means to detect and report an "exception circumstance" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:-

1:-Find the problem (*Hit the exception*).

2:-Inform that an error has occurred (*Throw the exception*).

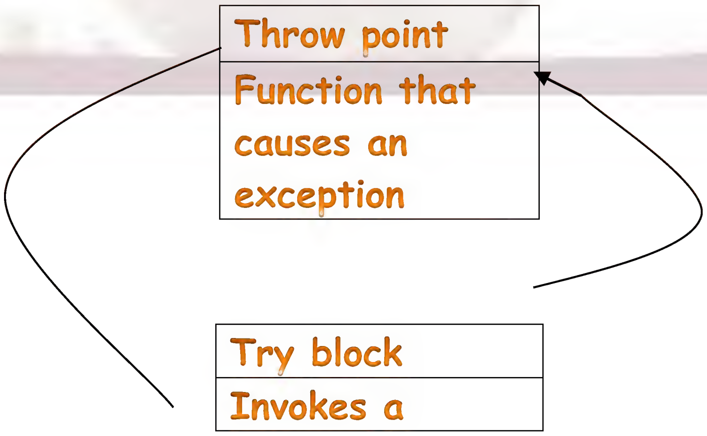
3:-Receive the error information (*Catch the exception*).

4:-Take corrective actions (*Handle the exception*).

EXCEPTION HANDLING MECHANISM

C++ exception handling mechanism is basically built upon three keywords, namely, try, throw, and catch.

The relationship in try, throw & catch:-



Throw point

Function that causes an exception

Try block

Invokes a

function that Contains an exception

Catch block

Catches and handles the exception

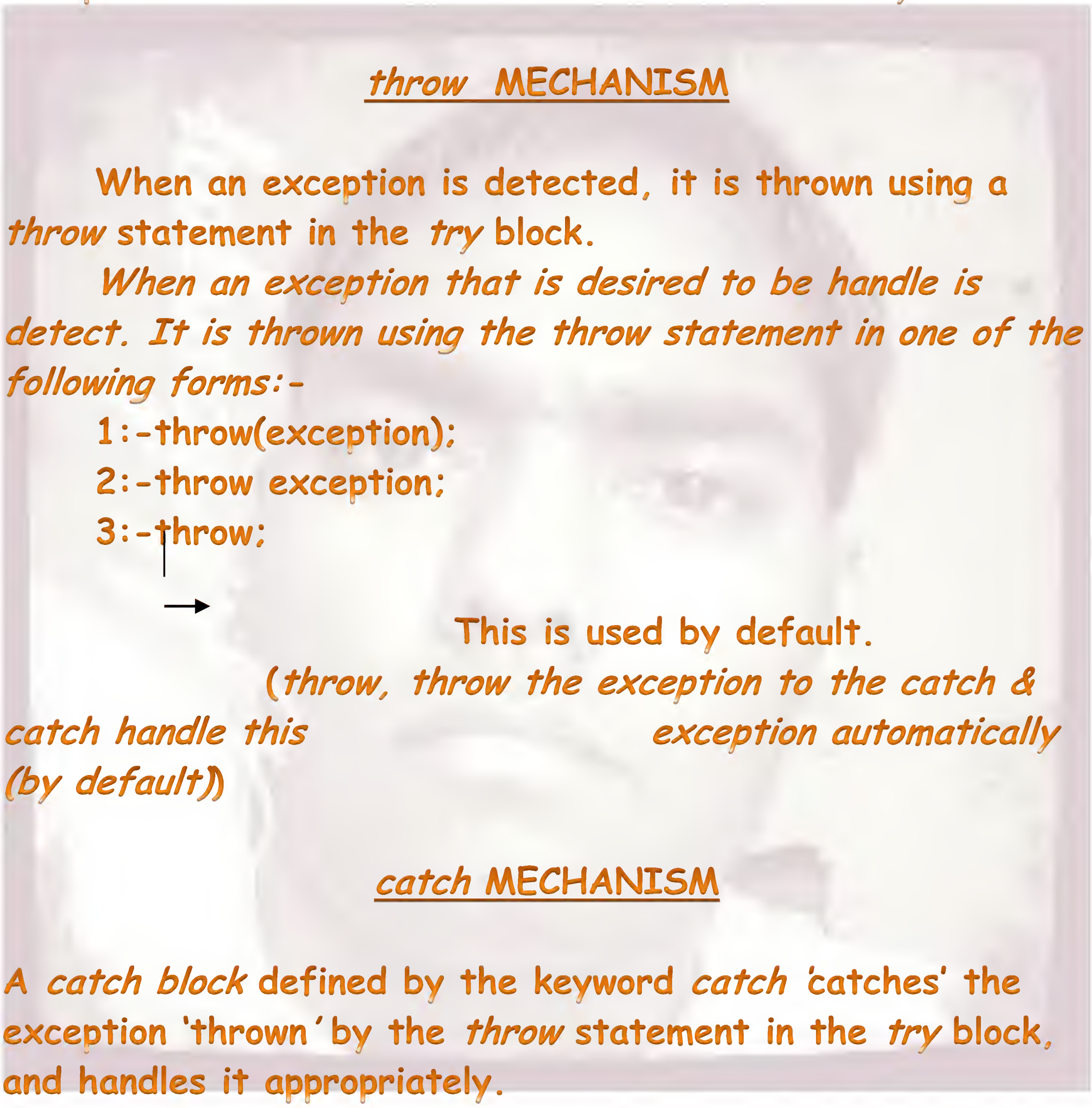
Standard format for exception handling:-

```
#include<iostream.h>
#include<conio.h>
        int main( )
        {
        -----
        ------
        try
        {
        ------
        ------
        throw(exception);
        -----
        -----
        }
        catch(type exception argument)
        {
        -----
        -----
        }
        ---
        ---
```

}

try MECHANISM

The keyword *try* is used to preface a block of statements (surrounded by braces) which may generated exceptions. This block of statement is known as *try* block.



throw MECHANISM

When an exception is detected, it is thrown using a *throw* statement in the *try* block.

When an exception that is desired to be handle is detect. It is thrown using the throw statement in one of the following forms:-

1:-throw(exception);
2:-throw exception;
3:-throw;

→ This is used by default.

(throw, throw the exception to the catch & catch handle this exception automatically (by default))

catch MECHANISM

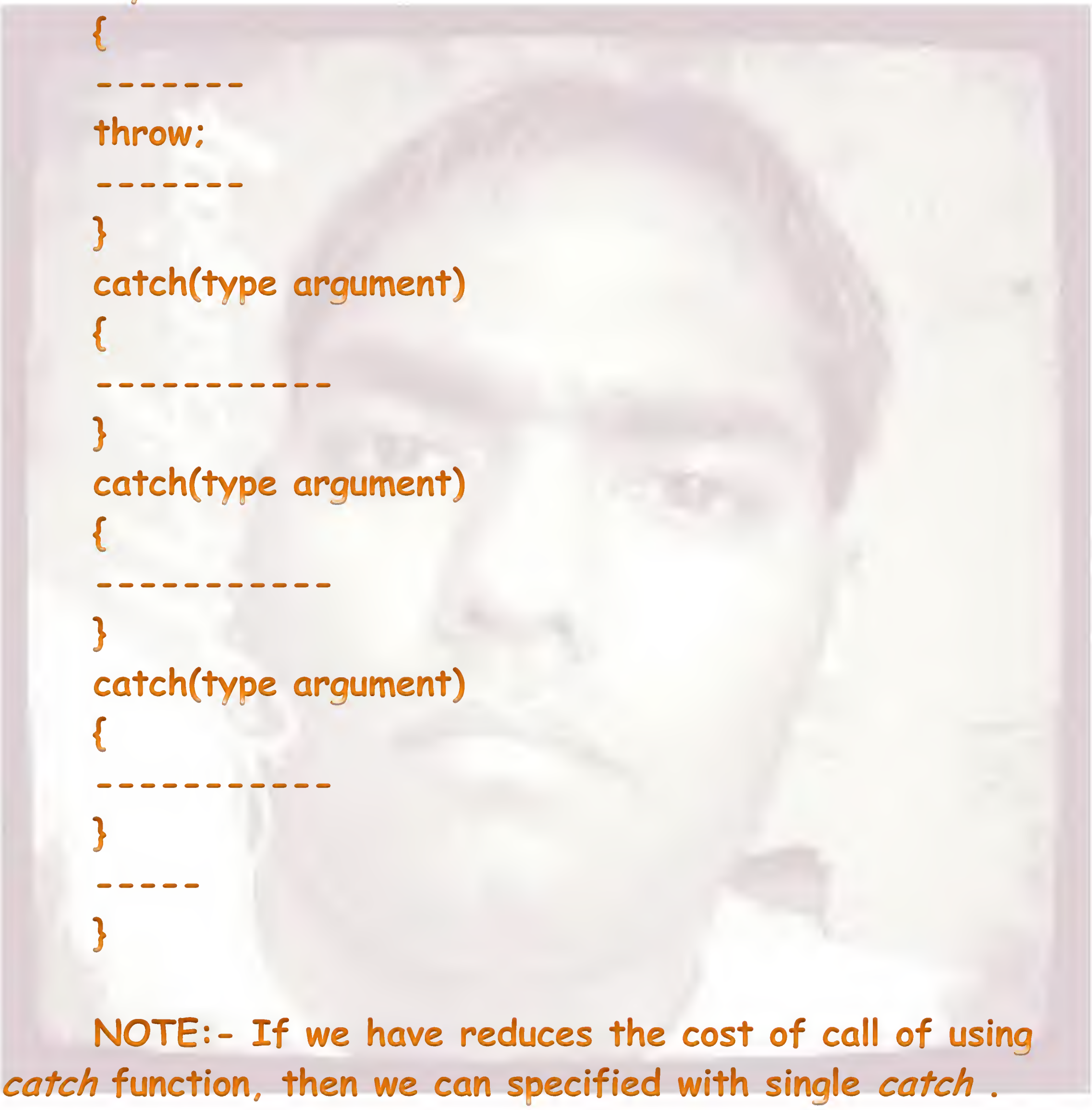
A catch block defined by the keyword *catch* 'catches' the exception 'thrown' by the *throw* statement in the *try* block, and handles it appropriately.

The catch statement catches an exception whose type matches with the type of the catch argument. When it is caught the code in the catch block is executed.

Multiple catch

One *try* block con hold multiple catch.

Standard format for multiple catch:-



```
#include<iostream.h>
#include<conio.h>
int main( )
{
-----------
try
{
-------
throw;
-------
}
catch(type argument)
{
-----------
}
catch(type argument)
{
-----------
}
catch(type argument)
{
-----------
}
-----
}
```

NOTE:- If we have reduces the cost of call of using *catch* function, then we can specified with single *catch* . Such as

```
Catch(-------)     //default catch//
{
-----
-----
}
```

RE-*throw* an exception

A handler (catch) may be decide to re-*throw* the exception caught without processing it. In such situation be may simply invoke *throw* without any argument as shown below:-

```
Throw;    //default throw//
```

MANAGING CONSOLE I/O OPERATIONS

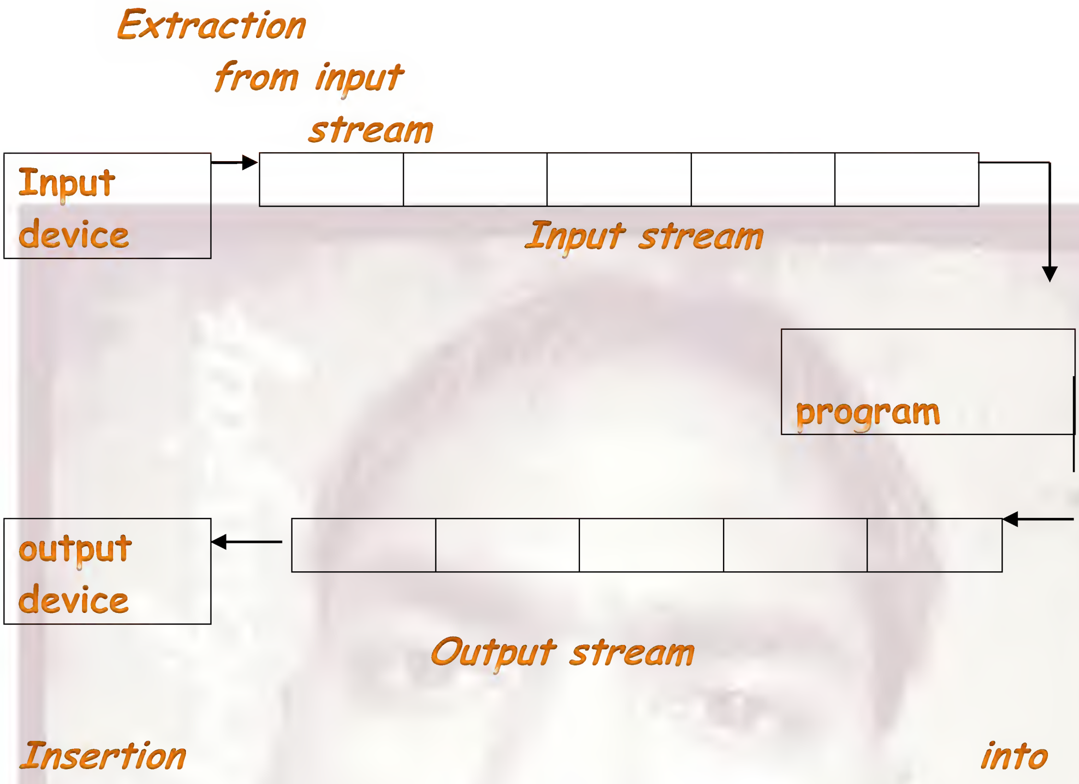
We can use any of them in the C++ programs, but we restrained from using then due to two reasons:-

1-I/O method in C++ support the concept of OOP.

2-I/O methods in C++ can not handle the user defined data type such as class objects.

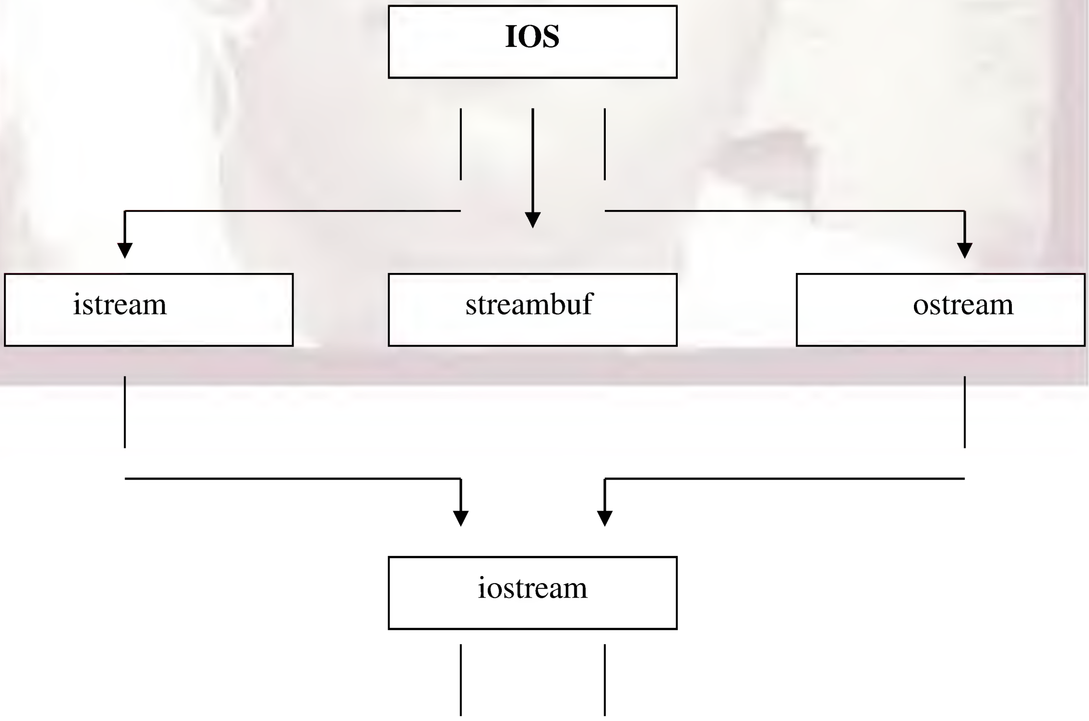
C++ uses the concepts of stream and stream classes to implement its I/O operations. How stream classes support the console oriented input output operations?

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called output stream.



Fig⇨Data streams

C++ STREAM CLASSES



Istream_withassign
Ostream_withassign

Fig.⇨stream classes for console I/O operations

1.IOS is the base class for istream (input stream) and ostream (output stream) which are in turn, base classes for iostream (input output stream).

Class name	Contents

2. The base class IOS is declared as a virtual base class so that only one copy of its members are inherited by the iostream.

3. the class IOS provide the basic support for the formatted and unformatted I/O operations.

4. The class istream provides the facilities for formatted and unformatted input.

5. The class ostream (through inheritance) provides the facilities for formatted and unformatted output.

Three classes ,namely, istream_with assign, ostream_with assign and iostream_with assign and assignment operator to these classes gives the details of these classes.

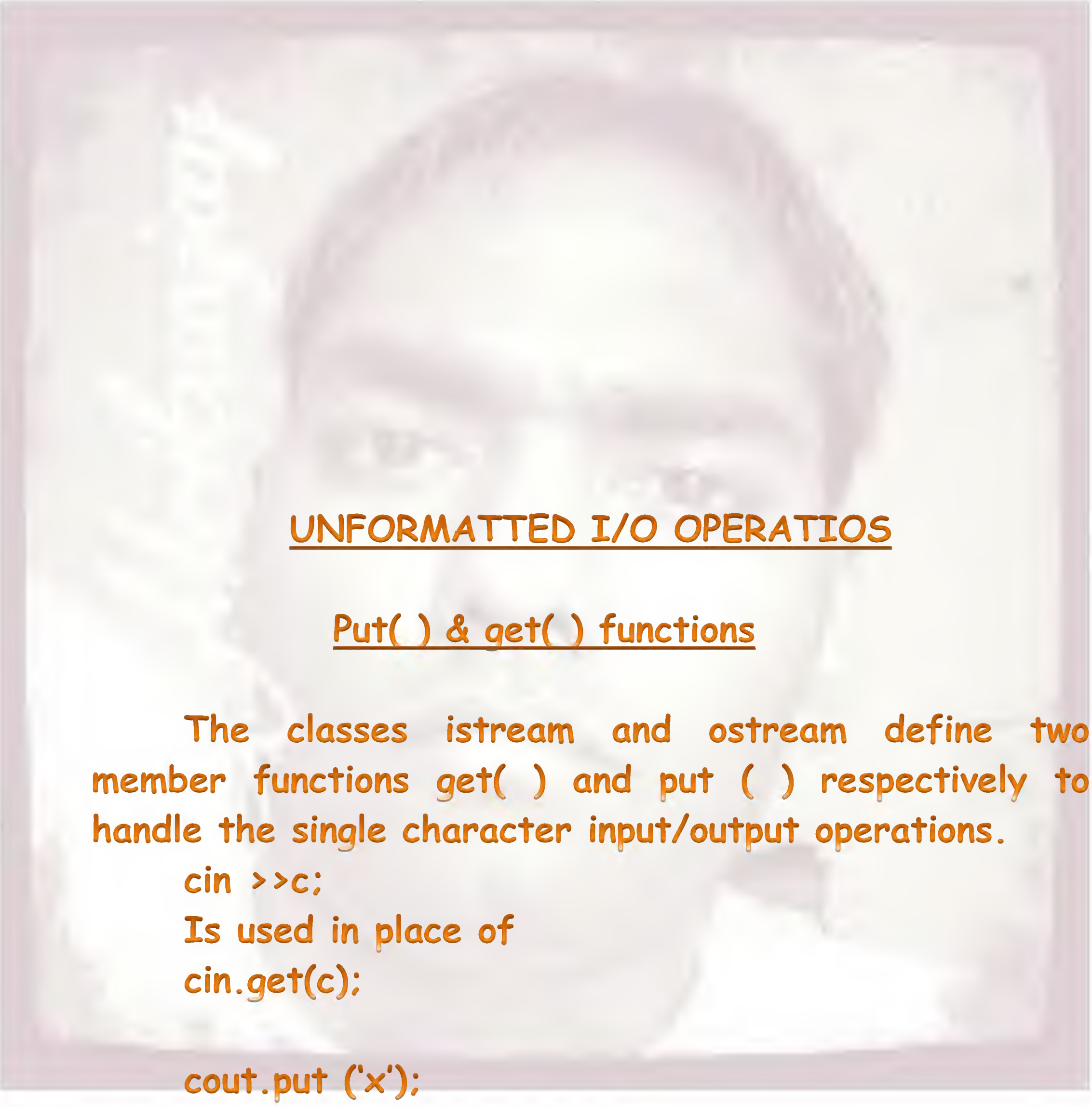
IOS (General input/output stream class) istream (input stream) ostream (output stream) iostream (input/output stream) streambuf	➧ Content basic facilities that are used by all other input and output classes. ➧ Also contains a pointer to buffer object (streambuf). ➧ Declares constants and functions that are necessary for handling formatted input and operations. ➧ Inherits the properties of IOS. ➧ Declares input functions such as get( ),getline( )and read( ) . ➧ Contains overloaded extraction operator >>. ➧ Inherits the properties of IOS . ➧ Declares output functions put( ) and write( ). ➧ Contains overloaded insertion operator <<. ➧ Inherits the properties of IOS

	istream and ostream through multiple inheritance and thus contains all the input an output functions. ➧ Provides an interface to physical devices through buffers. ➧ Acts as a base for filebuf class used IOS files.

Table⇨ stream classes console

for

operations



UNFORMATTED I/O OPERATIOS

Put( ) & get( ) functions

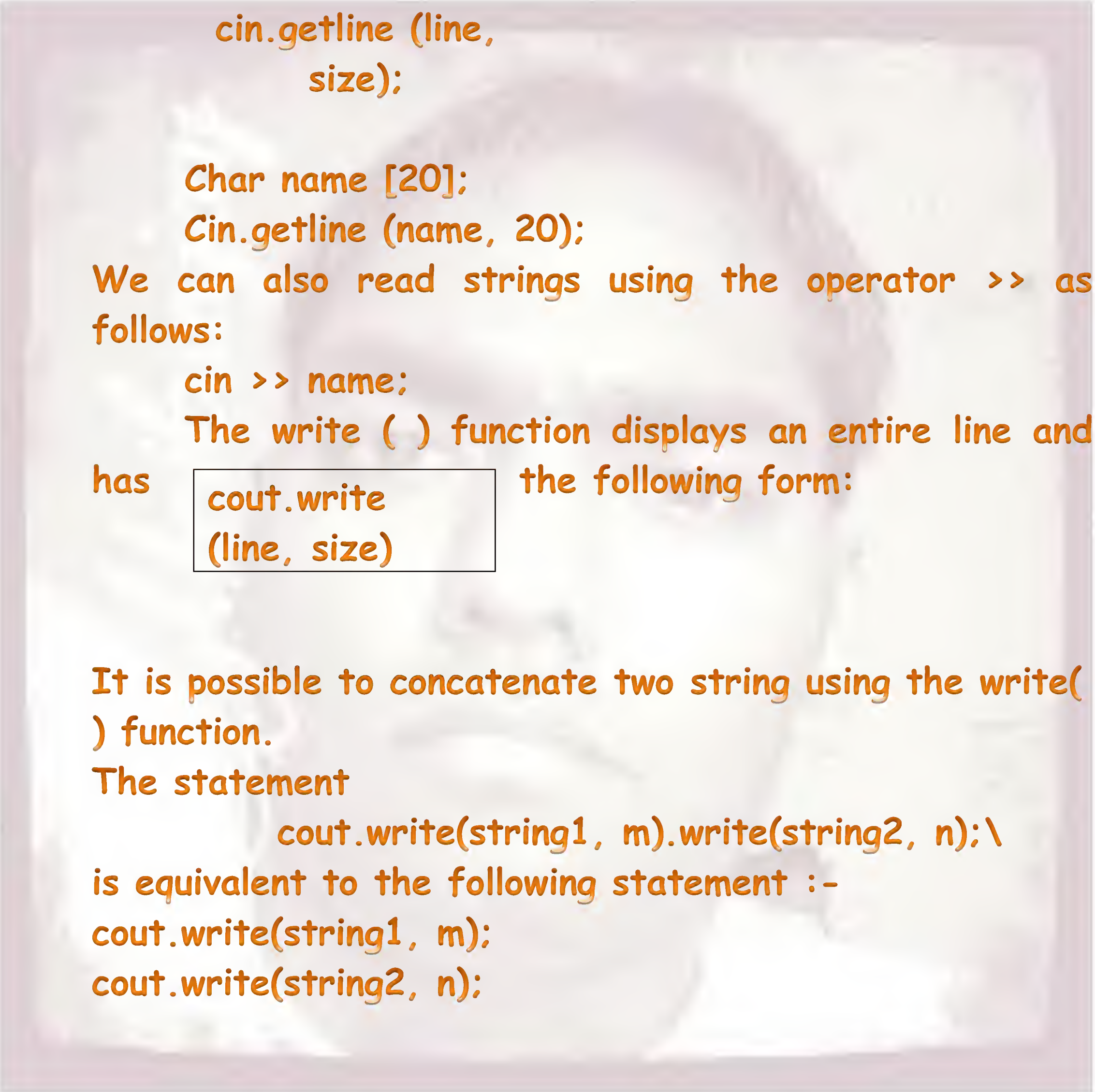
The classes istream and ostream define two member functions get( ) and put ( ) respectively to handle the single character input/output operations.

cin >>c;
Is used in place of
cin.get(c);

cout.put ('x');
cout<< 'x';
Display the character X and
cout.put (cn);

getline( ) & write ( ) functions

We can read and display a line of text more efficiently using the line oriented input/output functions getline( ) & write ( ). The getline ( ) function reads a whole line of text that ends with a newline character (transmitted by the RETURN key). This function can be invoked by using the object cin as follow:



```
        cin.getline (line,
              size);

      Char name [20];
      Cin.getline (name, 20);
```

We can also read strings using the operator >> as follows:

```
      cin >> name;
```

The write ( ) function displays an entire line and has the following form:

```
cout.write
(line, size)
```

It is possible to concatenate two string using the write( ) function.

The statement

```
            cout.write(string1, m).write(string2, n);\
```

is equivalent to the following statement :-

```
cout.write(string1, m);
cout.write(string2, n);
```

FORMATTED CONSOLE I/O OPERATIONS

C++ supports a number of features that could be used for formatting the output. these features include:-

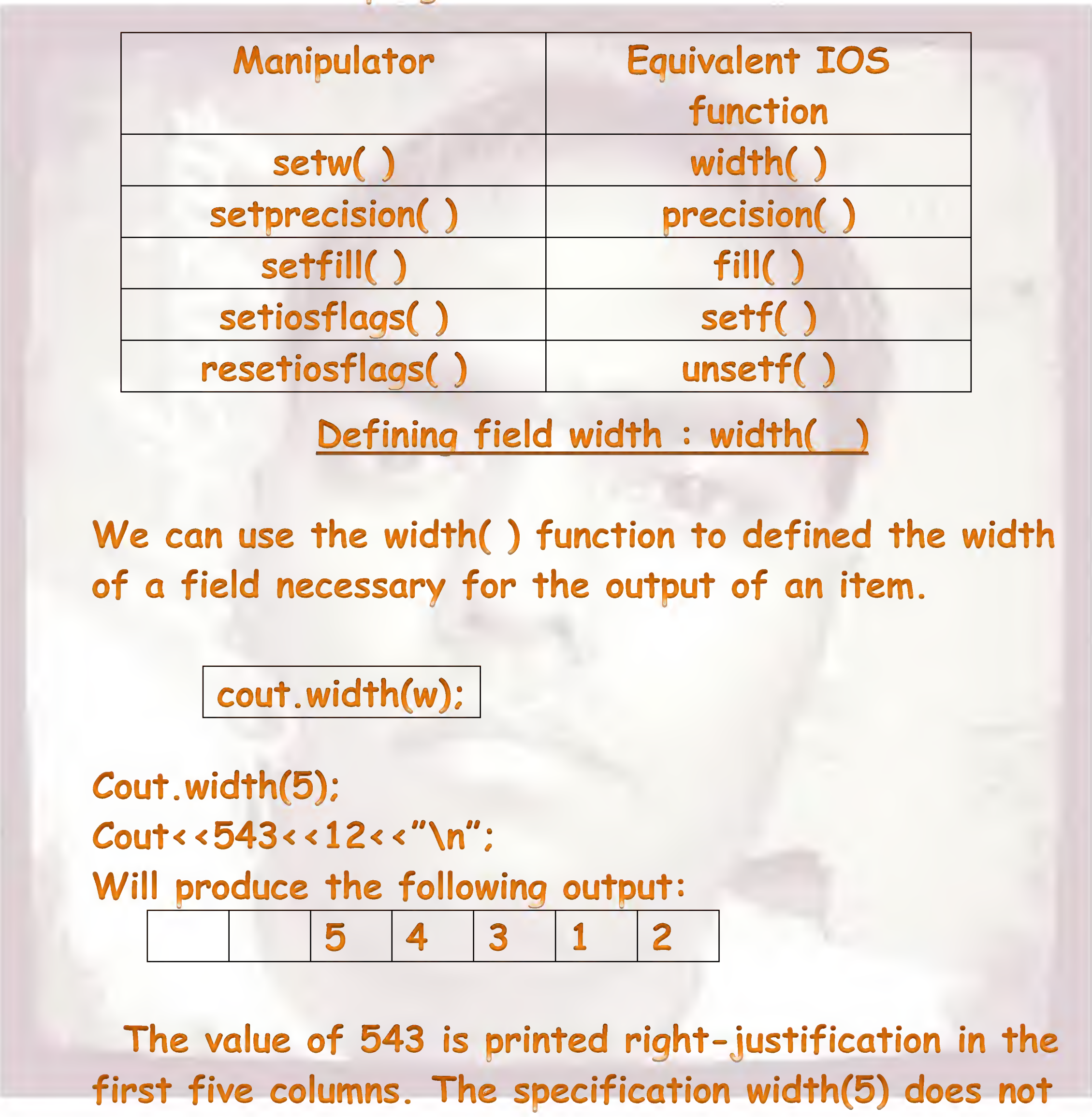
- IOS class functions and flags.
- Manipulators.
- User-defined output functions.

The IOS class contains a large number of member functions that would help us to format the output in the number of ways. The most important once among them are listed in table:-

Table⇨IOS format functions

Function	Task
width( )	➧ To specify the required field size of displaying an output value.
precision( )	➧ to specify the number of digits to be displayed after the decimal point of a float value.
fill( )	➧ to specify a character that is used to fill the unused portion of a field.
setf( )	➧ to specify format flags that can control the form of the output display (such as left-justifications and right-justifications).
unsetf( )	➧ To clear the flags specified.

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Table below shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.



Manipulator	Equivalent IOS function
setw( )	width( )
setprecision( )	precision( )
setfill( )	fill( )
setiosflags( )	setf( )
resetiosflags( )	unsetf( )

Defining field width : width( )

We can use the width( ) function to defined the width of a field necessary for the output of an item.

```
cout.width(w);
```

```
Cout.width(5);
Cout<<543<<12<<"\n";
```

Will produce the following output:

		5	4	3	1	2

The value of 543 is printed right-justification in the first five columns. The specification width(5) does not retain the setting for printing the number 12. this can be improved as followings:-

```
Cout.width(5);
Cout<<543;
Cout<<width(5);
```

```
Cout<<12<<"\n";
```

This produces the following output:

		5	4	3				1	2

Setting precision : precision( )

The floating numbers are printed with six digits after the decimal point. However we can specify the number of digits too be displayed after the decimal point while printing the floating-point numbers. This can be done by using the precision( ) member function as follows:-

```
cout.precision(d);
```

Where d is the number of digit to the right of the decimal. For example, the statements:

```
cout. precision(3);
cout<< sqrt(2)<< "\n";
cout<< 3.14159<< "\n";
cout<<2.50032<< "\n";
```

Will produce the following output:

```
1.141      (truncated)
3.142      (rounded to the nearest sent)
2.500      (no trailing zeros)
```

we can set different values to different precision as follows:

```
          cout. precision(3);
          cout<< sqrt(2)<< "\n";
          cout. precision(5);          // Reset the
precision
          cout<<3.14159<< "\n";
```

we can also combine the field specification with the precision setting. Example:

```
cout. precision(2);
cout.width (5);
cout<< 1.2345;
```

Filling and Padding: fill ( )

We can use the fill ( ) function to fill the unused positions by any desired character. It is used in the following form:

```
Cout.fill
(ch);
```

Where ch represents the character which is used for filling the unused positions. Example:

```
cout.fill ("*");
cout.width (10);
cout<<5250<< "\n";
```

The output would be:

*	*	*	*	*	*	5	2	5	0

Formatting Flags, Bit-field and setf ( ):

The setf ( ), a member function of the ios class, can provide answers to these any many other formatting question. The setf ( ) (*setf* stand for set flag)function can de used as follows:

```
cout.setf
(arg1, arg2)
```

The *arg1* is one of the formatting flags defined in the class *ios*. The formatting flags specify the flag format action required for the output. Another *ios* constant, *arg2*, is known as bit field specifies the group to which the formatting flag belongs.

Consider the following segment of code:

```
cout.fill ('x');
cout.setf (ios::left, ios::adjustfield);
cout.width (15);
cout<< "TABLE1"<< "\n";
```

Table: Flags and bit field for Setf( ) function.

Format Requried	Flag (arg1)	Bit-field(arg2)
Left-justified output Right-justified output Padding after sign or base Indicator (like+##20)	ios::left ios::right ios::internal	ios::adjustfield ios::adjustfield ios::adjustfield
Scientific Notation Fixed point notation	ios::scientific ios::fixed	ios::floatfield ios::floatfield
Decimal base Octal base Hexadecimal base	ios::left ios::right ios::internal	ios::basefield ios::basefield ios::basefield

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
        cout.fill ('*');
        cout.precision (3);
        cout.setf(ios::internal, ios::adjustfield);
        cout.setf(ios::scientific, ios:: floatfiled);
        cout.width (15);

        cout<< -12.34567<< "\n";
```

will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

Table⇨flags that do not have bit fields

Flag	Meaning
ios ::showbase	Use base indicator on output.
ios ::showpos	Print + before number.
ios ::showpoint	Show trialing decimal point and zero.
ios ::uppercase	Use uppercase letters for hex output.
ios ::skipus	Skip white space on input.
Ios ::unitbuf	Flush all stream after insertion.
Ios ::stdio	Flush stdout and

	stderr after insertion.

MANAGING OUTPUT WITH MANIPULATOR

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output formats. They provide the some features as that of the IOS member functions and flags. To access these manipulators, we must include the file iomanip in the program.

```
#include<iomanip.h>
```

Table⇨ Manipulators and meanings

Manipulator	Meaning	Equivalent
Setw(int w)	Set the field width to w.	Width( )
Setprecision(int d)	Set the floating point precision to d.	Precision( )
Setfill(int c)	Set the fill character to c.	Fill( )
Setiosflags(long f)	Set the format flag f.	Setf( )
Resetiosflags(long f)	Clear the flag specified by	Unsetf( )

	f.	
Endif	Insert new line and flush stream.	“\n”

Some example of manipulators are given below:-

```
Cout<<setw(10)<<12345;
```

This statement prints the value 12345 right-justified in a field width 10 character. The output can be made left -justified by modifying the statement as follows:-

```
Cout<<set(10)<<setiosflags(ios::left)<<12345;
```